חלק 5

מְנֻשָּׁקִים

הוראות לניקוי אקוואריום

How to clean your aquarium safely, by Sarah Davies

…

Turn off and remove all heaters and filters. These can be put in the sink and left until they are cleaned. Fill one of the new, clean buckets half full of water from the aquarium. Using the fish net, transfer the fish, one by one, to this bucket until all the fish are out of the aquarium. Next, remove all plants and ornaments. If the plants are living put them in the bucket with the fish. Put all the ornaments on the counter or in the bucket where the rocks go.

…

איזה דגים, קישוטים, ודליים בדיוק?

- כמובן שזה לא משנה; אותן ההוראות תקפות לאקוואריום עם דגי זהב ולאקוואריום עם ברקודות וכרישים, לדלי פח כחול ולדלי פלסטיק אדום
- ההוראות מתייחסות לעצמים באופן כללי שמאפשר שימוש בהוראות בהרבה מצבים שונים
- החוזה של הדלי: נוזלים ומוצקים שמכניסים לתוכו נשארים שם אם לא ממלאים אותו יותר מדי, מה שנכנס אפשר להוציא בדיוק באותו מצב (בפרט אסור שיהיו בדלי שיירי סבון)
- המימוש יכול להיות מפלסטיק, מפח, ואפילו מימושים כמו אגרטל חרסינה או סיר נירוסטה ממלאים את הדרישות

הקוד שלנו, עד עתה, לא היה כל כך כללי

- הלקוח שהשתמש בעצם מהמחלקה VersionedString עשה זאת דרך ייחוס מטיפוס VersionedString
- עצם מהמחלקה הזו מקיים כמובן את החוזה שהלקוח מסתמך עליו, אבל אולי יש עוד הרבה מחלקות שמקיימות את החוזה הזה
- למה שקוד הלקוח לא יוכל לפעול על כל מחלקה כזו?, למשל,

```
int Find(VersionedString vs, String s) {
  for (int i=0; i<vs.length(); i++)
    if (s.equals( vs.getVersion(i) ))
      return i;
}
```

זה פועל, אבל

- זה לא מאפשר להשתמש בשגרה הזו, המימוש הזה של אלגוריתם חיפוש, במצבים אחרים
- זה מקביל ל-"קח את דלי הפלסטיק האדום שמתחת לכיור (לא את דלי הספונג'ה הכחול, והעבר אליו את המים ואת דג הזהב"
- הפתרון: להפריד את הספק, המחלקה שמממשת את השירותים, מהחוזה, שאינו תלוי במימוש

מְנְשָׁקים (interfaces)

המנשק מגדיר את המנשקים של השירותים ואת החוזה

```
interface VersionedString {
  public void   add(String s)          ;
  requires … ; ensures: …
  public int    length()               ;
  requires … ; ensures: …
  public String getLastVersion()       ;
  requires … ; ensures: …
  public String getVersion(int i)      ;
  requires … ; ensures: …
}
```

ספק מממש מנשק

ספק שמממש מנשק מקיים את החוזה שלו; אין לו חוזה משלו

```
class LinkedVersionedString
      implements VersionedString {
  protected int       n;
  protected Version last;

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

הספק והחוזה

- הספק חייב לקיים את החוזה של המנשק שהוא מממש
- ככלל, אין לו חוזה משלו
- הספק חייב לספק שירות אם מצב התוכנית מקיים את תנאי הקדם של השירות
- אם מצב התוכנית מקיים את תנאי הקדם של שירות, אזי מצבה לאחר סיום השירות חייב לקיים את תנאי האחר
- אולי השירות יפעל גם במקרים שלא מקיימים את תנאי הקדם, ואולי השירות תמיד מותיר את התוכנית במצב טוב בהרבה מזה הנדרש בתנאי האחר
- אבל שירות טוב יותר מזה המובטח בחוזה אינו נדרש, וגם אם הספק מכריז על החוזה המשופר שלו, עדיף אולי ללקוח להימנע מלהסתמך עליו, כי אחרת לא יוכל להחליף ספק

ג'אווה לא מחפשת דליים מתחת לכיור

- אני כן, ואם הייתי מנסה לעקוב אחרי הוראות ניקוי האקווריום, והייתי קורא שצריך דלי נקי, הייתי מחפש ומוצא
- אבל ג'אווה לא, ולכן, הקוד הבא יכשל ולא יעבור קומפילציה,

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

- ג'אווה רוצה שהלקוח יגיד איזה סוג עצם (מאיזו מחלקה) הוא רוצה לבנות; ג'אווה לא תנחש עבורו, אפילו אם יש רק מחלקה אחת שמממשת את טיפוס המנשק של המשתנה, או רק מחלקה אחת שמממשת את המנשק ויש לה בנאי מתאים
- מה שצריך בג'אווה הוא

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

אסימטריה

- שגרה כמו Find שהגדרנו לחיפוש גרסה מסוימת של מחרוזת יכולה להשתמש רק בטיפוס המנשק, ולכן לעבוד עם כל מימוש שיועבר לה כארגומנט
- אבל קוד שצריך ליצור עצמים לא יכול להשתמש בהם רק דרך טיפוסי המנשק המתאימים, מכיוון שלאופרטור new חייבים להעביר שם של מחלקה, לא של מנשק

הפתרון: בתי חרושת (factories)

עצם שממש מנשק יצירת עצמים מטיפוס מנשק מסוים

```
interface VersionedStringFactory {
  public VersionedString construct();
}

class LinkedVersionedStringFactory
      implements VersionedStringFactory {
  public VersionedString construct() {
    return new LinkedVersionString();
  }
}
```

שימוש בבית חרושת (1)

```
class SomeClass {
 private VersionedStringFactory f;
 public SomeClass(VersionedStringFactory f,
                  ...)
   { this.f = f; ... }
 ...
 public someMethod() {
   VersionedString vs = f.construct();
   vs.add("First version");
   ...
 }
```

שימוש בבית חרושת (2)

- בזמן יצירת העצם ששירותיו צריכים VersionedString חדשים, מעבירים לבנאי שלו בית חרושת; אפשר להעביר לו בית חרושת שייצר LinkedVersionedString או בית חרושת שייצר עצמים אחרים שמקיימים את המנשק
- בקוד של המחלקה הלקוחה אין שום אזכור לספקים ספציפיים, ולכן היא יכולה לעבוד עם כל ספק, כולל ספק שייכתב בעתיד
- ניתן כמובן גם להעביר את בית החרושת ישירות לשירות שהקוד שלו צריך עצמים חדשים
- המבנה הרצוי תלוי בעיקר בשאלה מי הקוד שמחליט באיזה ספק להשתמש, ולכן באיזה בית חרושת להשתמש; בדרך כלל, זהו קוד בקרת תצורה שרחוק למדי מהקוד המשתמש

בתי חרושת משוכללים

בית חרושת יכול לייצר עצמים תוך שימוש בארגומנטים

```
interface VersionedStringFactory {

  public VersionedString construct();
```

Ensures: returns a reference to a new VersionedString

```
  public VersionedString construct(int m);
```

Ensures: returns a reference to a new VersionedString that keeps at least the m most recent versions

```
}
```

אם החוזה מרשה, מותר להתבטל

```
class LinkedVersionedStringFactory
      implements VersionedStringFactory {
  public VersionedString construct() {
    return new LinkedVersionString();
  }
```

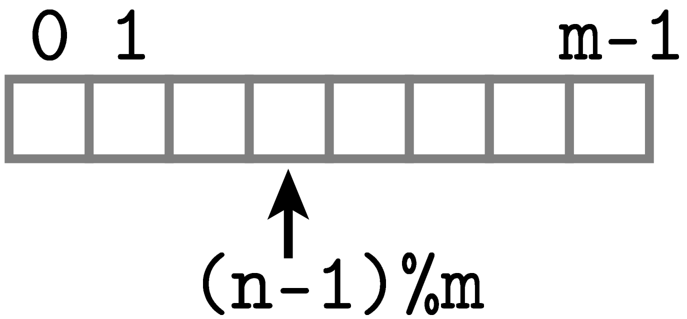
keeps an unlimited number, so satisfies the contract

```
  public VersionedString construct(int m) {
    return new LinkedVersionString();
  }
```

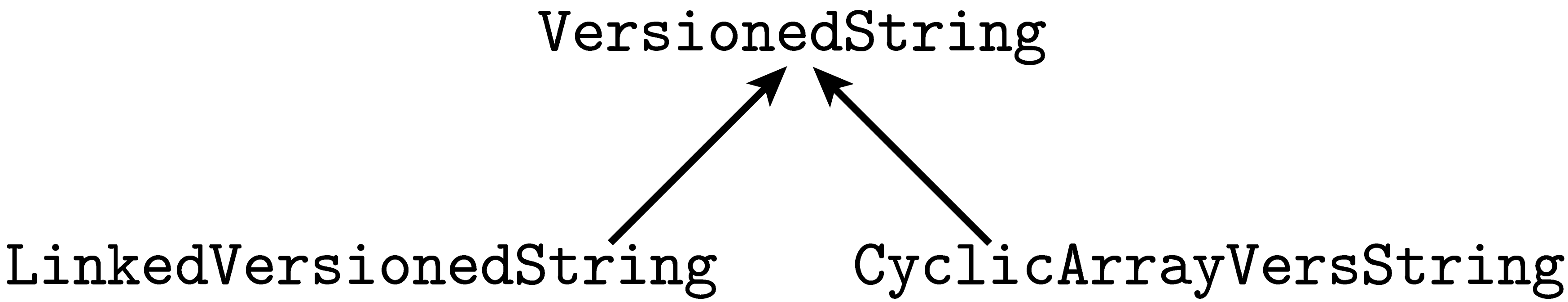
`construct(int)` מאפשר ללקוח לתת עצה לבית החרושת;
בית החרושת לא חייב לנצל את העצה

אבל אפשר להתאמץ

```
class SmartVersionedStringFactory
      implements VersionedStringFactory {
  public VersionedString construct() {
    return new LinkedVersionString();
  }
  public VersionedString construct(int m) {
     return new CyclicArrayVersString(m);
```



ניצני ארגון עבור טיפוסים



- בין שלושת הטיפוסים יש יחסים: שתי המחלקות מממשות את המנשק
- המנשק יותר כללי מהמחלקות שמממשות אותו
- משתנה או שדה מטיפוס כללי יותר (כאן מנשק) יכול להתייחס לעצם מטיפוס יותר ספציפי
- אבל לא להיפך

עוד דוגמה: מיון מערך

requires: a != null && a's elements != null

ensures: a becomes sorted

```
void insertionSort(Comparable a[]) {
  int i,j;
  for (j=1; j<a.length; j++) {
    Comparable key = a[j];
    for (i=j-1;
         i>=0 && a[i].compareTo(a[j])>0;
         i--) a[i+1] = a[i];
    a[i+1] = key;
  } }
```

מנשק להשוואות

```
interface Comparable {
  requires: other != null
  ensures: return == 0 iff this = other
           return == 1 iff this > other
           return ==-1 iff this < other
  int compareTo(Comparable other);
}
```

בספריות של השפה מוגדר מנשק java.lang.Comparable

הוא דומה ברוחו ובמהותו למנשק שהגדרנו כאן, אבל לא זהה לו לגמרי; בהמשך נבין למה

מימוש המנשק ב-VersionedString

```
class LinkedVersionedString
      implements VersionedString,
                 Comparable {
  protected int      n;
  protected Version last;

  int compareTo(Comparable other) {
    if (this.n > other.n) return  1;
    if (this.n < other.n) return -1;
    return 0;
  }
```

לפעמים אני מרצה ולפעמים הורה

- וזה נורמלי
- אני מספק שירותים אחרים בתור מרצה ואחרים בתור הורה
- אבל אני תמיד נשאר אני (ומממש שני מנשקים שונים)
- יש מכונת פקס שלפעמים היא פקס, לפעמים מכונת צילום, לפעמים טלפון, ולפעמים משיבון
- הגדרה של מחלקה יכולה להצהיר שהיא מממשת מספר מנשקים

אבל המימוש לא נכון!

```
int compareTo(Comparable other) {
  if (this.n > other.n) return  1;
  if (this.n < other.n) return -1;
  return 0;
}
```

- השירות למעשה מניח ש-other הוא מטיפוס LinkedVersionString
- אבל other הוא מטיפוס Comparable, לא מטיפוס LinkedVersionString, ולכן הקומפיילר לא ירשה להתייחס לשדה n דרכו

ניסיון לפתרון הבעייה

נדרוש בתנאי הקדם של insertionSort שכל העצמים במערך יהיו מאותה מחלקה, ונתייחס ל-other בהתאם:

```
int compareTo(Comparable other) {
 LinkedVersiondString other_lvs
   = (LinkedVersiondString) other;
 if (this.n > other_lvs.n) return  1;
 if (this.n < other_lvs.n) return -1;
 return 0;
}
```

האופרטור שבסוגריים נקרא יציקה (cast), והוא מייצר ייחוס מטיפוס נתון לעצם נתון, אם העצם מתאים לטיפוס

יציקות (casts)

- יציקה מצליחה אם הייחוס שיוצקים מתייחס לעצם מתאים לטיפוס שיוצקים לתוכו
- יציקה למטה (downcast): יציקה של ייחוס לטיפוס פחות כללי; כרגע הכוונה ליציקה של ייחוס למנשק לייחוס למחלקה שמממשת את המנשק; בהמשך נראה שיש עוד מקרים
- יציקה למעלה (upcast): יציקה של ייחוס לטיפוס יותר כללי, למשל יציקה של ייחוס למחלקה לייחוס למנשק שהמחלקה מממשת; שוב, בהמשך נראה עוד מקרים
- יציקה למעלה תמיד מצליחה, ולא מצריכה אופרטור מפורש; היא פשוט גורמת לקומפיילר לאבד מידע
- יציקה למטה עלולה להיכשל; בהמשך נראה מה קורה אז
- אפשר לצקת גם משתנים ושדות פרימיטיביים

אבל היציקה לא פתרה את הבעיה

- כי הלקוח לא בהכרח יודע אם כל העצמים שהוא רוצה למיין שייכים לאותה מחלקה או לא
- למשל, אם `SmartVersionedStringFactory` ייצר את העצמים, יתכן שהלקוח קיבל עצמים מכמה מחלקות
- אם בעיני הלקוח אפשר למיין עצמים שמממשים `VersionedString`, אז סימן שהמיון צריך לבטא תכונות של העצמים שהלקוח מודע להם, לא את המימוש הנסתר
- אי לכך, צריך להצהיר שכל עצם שמממש `VersionedString` מממש גם `Comparable`

מנשק מבטיח לממש מנשק אחר

```
interface VersionedString
          extends Comparable {
  public void   add(String s)       ;
  public int    length()            ;
  public String getLastVersion()    ;
  public String getVersion(int i)   ;
}
class LinkedVersionedString
      implements VersionedString { ... }
```

- המנשק לא צריך להצהיר על compareTo, קיום השירות מובטח מעצם העובדה שהמנשק מרחיב את Comparable

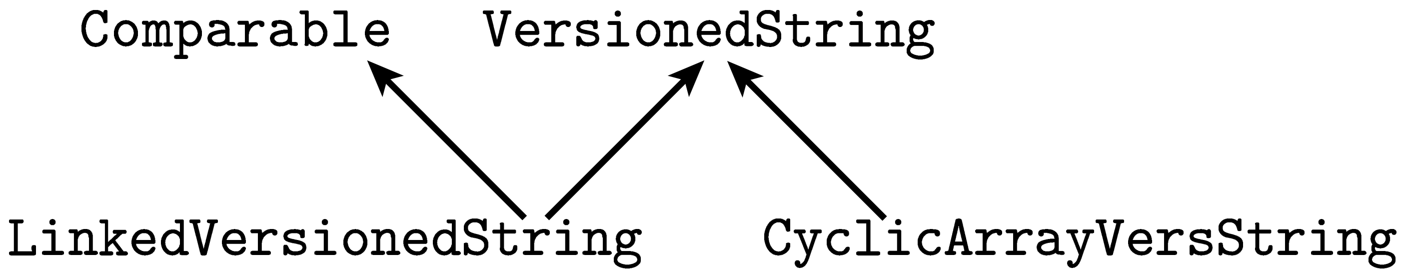
והמימוש הנכון...

```
int compareTo(Comparable other) {
  VersiondString other_vs
    = (VersiondString) other;
  if (this.length() > other_vs.length())
    return  1;
  if (this.length() < other_vs.length())
    return -1;
  return 0;
}
```

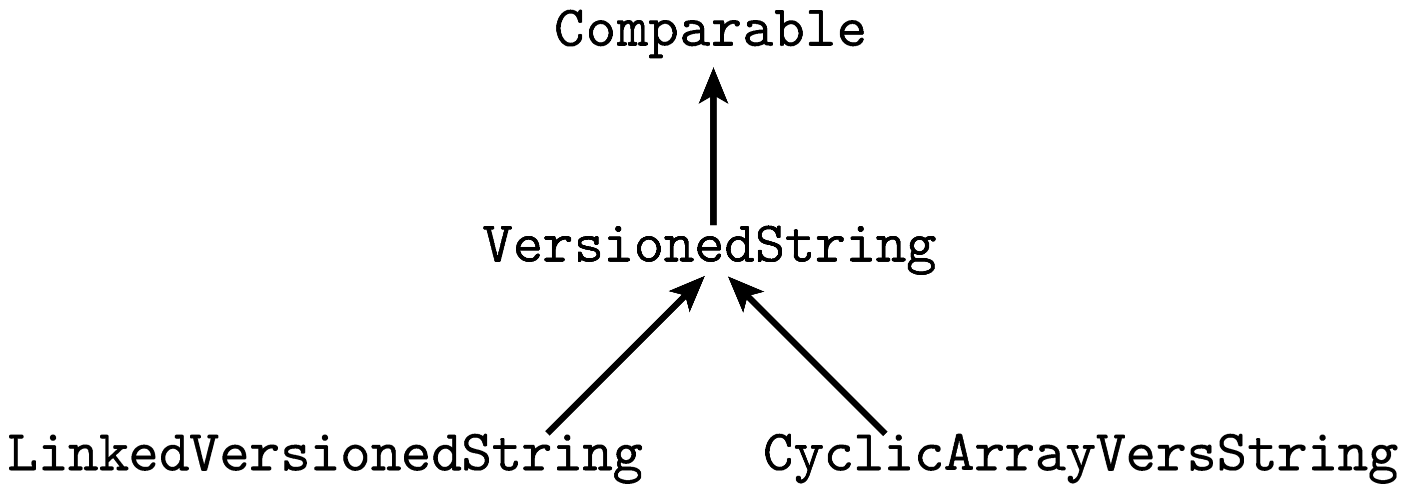
קצת פגום: למרות שהמימוש לא ספציפי למחלקה צריך
לשכפלו בכל מימוש של VersionedString; בהמשך נתקן

היררכית הטיפוסים גדלה

זה לא היה
מבנה מוצלח
במקרה זה,
אבל הוא
הראה
שהיחסים בין טיפוסים הם גרף א-ציקלי מכוון



המבנה הזה היה יותר מוצלח, והוא מראה שגרף היחסים עשוי להיות עמוק



סיכום מנשקים

- שימוש בטיפוס מנשק מאפשר ללקוח להצהיר שייחוס (משתנה או שדה) מתייחס לעצם שמספק שירותים מסוימים, בלי לציין מאיזו מחלקה העצם; זה מאפשר כלליות בלקוח
- לקוח כזה נקרא רב-צורתי (polymorphic)
- שימוש בבית חרושת (factory) מאפשר ללקוח לבנות עצמים עם מנשק מסוים בלי לציין בעצמו מאיזו מחלקה הם
- מחלקה יכולה לממש מספר מנשקים
- מנשק יכול להרחיב מנשק אחד או מספר מנשקים אחרים
- רב צורתיות מאפשרת ניצול של קוד קיים במקרים נוספים ומונעת שכפול של קוד, שכפול שהוא יקר לפיתוח ותחזוקה

חלק 6
חריגים

החוזים שהגדרנו אינם סימטריים

- אם הלקוח רוצה שתנאי האחר יתקיים, הוא צריך להבטיח שתנאי הקדם מתקיים
- אם תנאי הקדם אינו מתקיים, הלקוח אינו רשאי להניח מאומה לגבי פעולת השירות, אפילו לא שיסתיים
- מכאן שאם הלקוח אינו מצליח לקיים את תנאי הקדם, אין לו טעם בכלל לקרוא לשירות; הוא יכול לוותר על השירות, או לנסות מאוחר יותר שוב, או לנסות להשיג את קיום תנאי האחר בדרך אחרת, אבל אין טעם לקרוא לשירות
- אבל אם הספק אינו מצליח לקיים את תנאי האחר, אין לו אפשרות לבטל את הקריאה לשירות: היא כבר התבצעה
- הספק יכול לקיים את חלקו, או להשתמט, אבל אינו יכול לבטל את העסקה

למה שהספק יכשל?

- הרי הכוונה הייתה שתנאי הקדם יהיה מספיק לקיום תנאי האחר על ידי הספק ושאפשר יהיה להוכיח נכונות הספק
- אבל לפעמים כדאי להגדיר תנאי קדם חלש יותר שאינו מספיק, שלעצמו, להבטחת יכולת הספק לקיים את תנאי האחר
- במקרים כאלה, משמעות הקריאה לשירות היא: אני (הלקוח) ביצעתי את המוטל עלי (תנאי הקדם); כעת נָסֶה אתה (הספק) לבצע עבורי את השירות, והודע לי אם תכשל
- יש שתי סיבות טובות להגדיר תנאי קדם חלש כזה
- ועוד סיבה נפוצה אבל לא טובה, שגם אותה נסביר

סיבה טובה ראשונה: חוסר שליטה

```
import java.io.*;
...
  File f  = new File("A:\config.dat");
```
f represents the file's name; may or may not exist
```
  if ( f.exists() ) {
    FileInputStream is
      = new FileInputStream(f);
```
now access the file

הניסיון להבטיח שהקובץ קיים, בעזרת השאילתה exists,
לפני שפותחים וניגשים אליו שגוי: אולי הוא נמחק בינתיים

חוסר שליטה בגלל בו זמניות

- הדוגמה הזו משקפת את העובדה שהעצמים הרלוונטיים לביצוע מוצלח של השירות, כאן קובץ, אינם בשליטה מוחלטת של הלקוח שקורא לשירות
- גם אם הלקוח מוודא שהקובץ קיים לפני הקריאה לשירות, עדיין יתכן שהוא ימחק בין הוידוא ובין הקריאה לשירות, על ידי תוכנית אחרת, אולי של משתמש אחר
- ואולי הקובץ ימחק על ידי חוט (thread, תהליכון) של אותה תוכנית, אם יש לה כמה חוטים
- הבעיה הבסיסית היא חוסר שליטה מוחלטת בעצמים הרלוונטיים; לעוד מישהו יש שליטה עליהם, שליטה מספיקה על מנת להעביר אותם למצב שאינו מאפשר לספק לפעול
- ולכן הלקוח אינו יכול להבטיח שהספק מסוגל להצליח

איך לבקש מהספק לנסות

```
import java.io.*;
...
  File f  = new File("A:\config.dat");
  try {
    FileInputStream is
      = new FileInputStream(f);
    access the file (but only if the constructor succeeds)
  } catch (FileNotFoundException fnfe) {
    how to act if f does not exist
  }
```

אם הספק נכשל, התוכנית עוברת מייד לגוש ה-catch

טיפול בחריגים בג'אווה (1)

- חריג יכול להיזרק ע"י פקודת `throw` (נראה בהמשך).
- פקודת `throw` גורמת להפסקת הביצוע הרגיל, והמשערך מחפש `exception handler` שיתפוס את החריג.
- `exception handler` נכתב כמשפט try/catch/finally.
- אם הבלוק העוטף מכיל טיפול בחריג זה, קטע הטיפול מתבצע, ולאחריו עוברים לבצע את הקוד שאחרי הבלוק.
- אם אין טיפול בחריג הזה בבלוק הנוכחי, המשערך מחפש `handler` בבלוק העוטף, או בקוד שקרא לשרות הנוכחי.
- החריג מועבר במעלה מחסנית הקריאות. אם גם ב main אין טיפול, תודפס הודעה וביצוע התכנית יסתיים.

טיפול בחריגים בג'אווה (2)

```
try {
      main code block
   } catch (Exception1 exc) {
    when Exception1 thrown inside try, transfer here
   } catch (Exception2 exc) {
    when Exception2 thrown inside try, transfer here
   } finally {
  optional part. Executed no matter how the try block
  is exited.  }
```

- קטע הקוד של finally, אם קיים, יתבצע בכל יציאה מהבלוק, בין אם היה חריג או לא, ובין אם טופל כאן או לא.

חוסר שליטה בגלל פרוטוקולים

- שתי תוכניות (אולי על מחשבים שונים) מנהלות דו-שיח בפרוטוקול מובנה, למשל דפדפן ושרת http
- בכל אחת מהן הקשר מיוצג בעזרת עצם; בדפדפן ג׳אווה, למשל, הקשר מיוצג בלקוח על ידי עצם מהמחלקה `java.net.HttpURLConnection`
- גם אם הלקוח של העצם הזה ימלא את חלקו בחוזה בקפדנות, עדיין יתכן שהצד השני בקשר (השרת) לא יתנהג בדיוק לפי הפרוטוקול
- קורה במשפחות הכי טובות (שמישהו לא מתנהג לפי הפרוטוקול)
- העצם מושפע מהעולם החיצון (מהשרת) ולכן ללקוח של העצם אין שליטה מלאה עליו

סיבה טובה שנייה: קושי לבדוק את התנאי

```
Matrix a = ...;
Vector b = ...;
Vector x;
```

Matrix.solve requires nonsingularity

```
if ( a.nonsingular() )
  x = a.solve(b);
```

solves Ax=b

- חוזה אלגנטי אבל לא יעיל להחריד: הבדיקה האם מטריצה A הפיכה יקרה בערך כמו פתרון מערכת המשוואות Ax=b
- עדיף לבקש מהעצם לנסות לפתור את המערכת, ושיודיע לנו אם הוא נכשל בגלל שהמטריצה לא הפיכה

מחויבויותיו של ספק שנכשל

- שירות שמסתיים בהצלחה חייב לקיים את תנאי האחר ואת המשתמר של המחלקה
- תנאי האחר דרוש ללקוח
- קיום המשתמר מאפשר לשירותים אחרים שהעצם יספק בעתיד לפעול
- מה נדרש משירות שנכשל?
- ראינו כבר שהוא חייב להודיע ללקוח על הכישלון, כדי שהלקוח לא יניח שתנאי האחר מתקיים; בדרך כלל, גוש ה-`try` בלקוח מפסיק לפעול וגוש ה-`catch` מופעל
- ברור שהשירות שנכשל לא חייב לקיים את תנאי האחר
- האם השירות שנכשל צריך לשחזר את המשתמר?

כמובן שהשירות צריך לשחזר את המשתמר

- מכיוון שהעצם ממשיך להתקיים, ויתכן ששירותים אחרים שלו יקראו בעתיד
- שירותים אחרים צריכים למצוא את העצם במצב שמאפשר להם לפעול
- ברור שעדיף להחזיר את העצם למצב שבו שירותים אחרים יוכלו לא רק לפעול, אלא גם להצליח
- אבל אולי העצם במצב גרוע כל כך שכל שירות שיופעל בעתיד יכשל גם הוא, אבל השירותים העתידיים צריכים לפחות לפעול ולדווח ללקוחות שלהם על כישלון

הוכחת נכונות של ספק

- הלקוח צריך לקיים תנאי קדם, מועיל אבל אולי לא מספיק
- השירותים השונים של העצם צריכים לדאוג לקיום המשתמר, בין אם הם הצליחו ובין אם לא
- אם מתקיימים תנאי הקדם והמשתמר, שירות חייב להסתיים
- אם בנוסף מתקיים תנאי צד מסוים, השירות מצליח
- אם תנאי הצד לא מתקיים, השירות נכשל ומודיע על חריג (throws an exception)

precondition & invariant

& side-condition → invariant & postcondition

precondition & invariant

& not side-condition → invariant & exception is thrown

תנאי הצד

- החוזה לא חייב להגדיר בדיוק את תנאי הצד שמונע חריג
- תנאי הצד הזה יכול להיות קשה להבעה ו/או לחישוב
- הלקוח ממילא אינו אחראי לקיום תנאי הצד
- אבל הגדרה של תנאי הצד, או לפחות הגדרה של תנאי מספיק למניעת חריג, יכולה לסייע לתוכניתן/נית להימנע מחריג או לפחות להבין למה הוא קורה
- למשל, יש מקרים שבהם אפשר לדעת מראש שמטריצה הפיכה, כמו משולשית בלי אפסים על האלכסון

מה עושה לקוח שמקבל חריג?

```
int compareTo(Comparable other) {
  VersiondString other_vs;
  other_vs = (VersiondString) other;
  if (this.length() > other_vs.length())...
```

יציקה למטה עלולה להודיע על חריג אם העצם (other) אינו מטיפוס שמתאים לניסיון היציקה (כאן VersionedString)

אם הלקוח לא מטפל בחריג, כמו כאן (לא התייחסנו כלל לאפשרות של חריג), קוד הלקוח מפסיק לרוץ ומודיע למי שקרא לו על החריג

זה הגיוני: הלקוח הניח שיקבל שירות מסוים, השירות נכשל, הלקוח לא יכול לקיים את תנאי האחר שלו עצמו

שיפור קטן

```
int compareTo(Comparable other) {
  VersiondString other_vs;
  try {
    other_vs = (VersiondString) other; }
  catch (java.lang.ClassCastException ce) {
    throw new IncomparableException();
  }
  if (this.length() > other_vs.length())
  ...
```

הלקוח יכול לתרגם את ההודעה כך שתהיה מובנת ללקוח שלו: מי שקרא ל-compareTo לא ביקש לצקת אלא להשוות

היחלצות מצרה

```
Matrix a = ...;
Vector b = ...;
Vector x;
try {
  x = a.solve(b);   solves Ax=b, fast algorithm
} catch (CloseToSingularException ctse) {
  x = a.accurateSolve(b);   try harder
}
```

לפעמים הלקוח יכול למָסֵך חריג, למשל על ידי שימוש בדרך אחרת, אולי יקרה יותר, לביצוע השירות

מה קורה אם המטריצה בדיוק סינגולרית והניסיון השני נכשל?

עוד דוגמה להיחלצות מצרה

```
FileInputStream is;
try {
 is = new FileInputStream("A:\config.dat");
} catch (FileNotFoundException fnfe) {
 is = new FileInputStream("A:\config");
}
```

access the file (but only if the input stream was created)

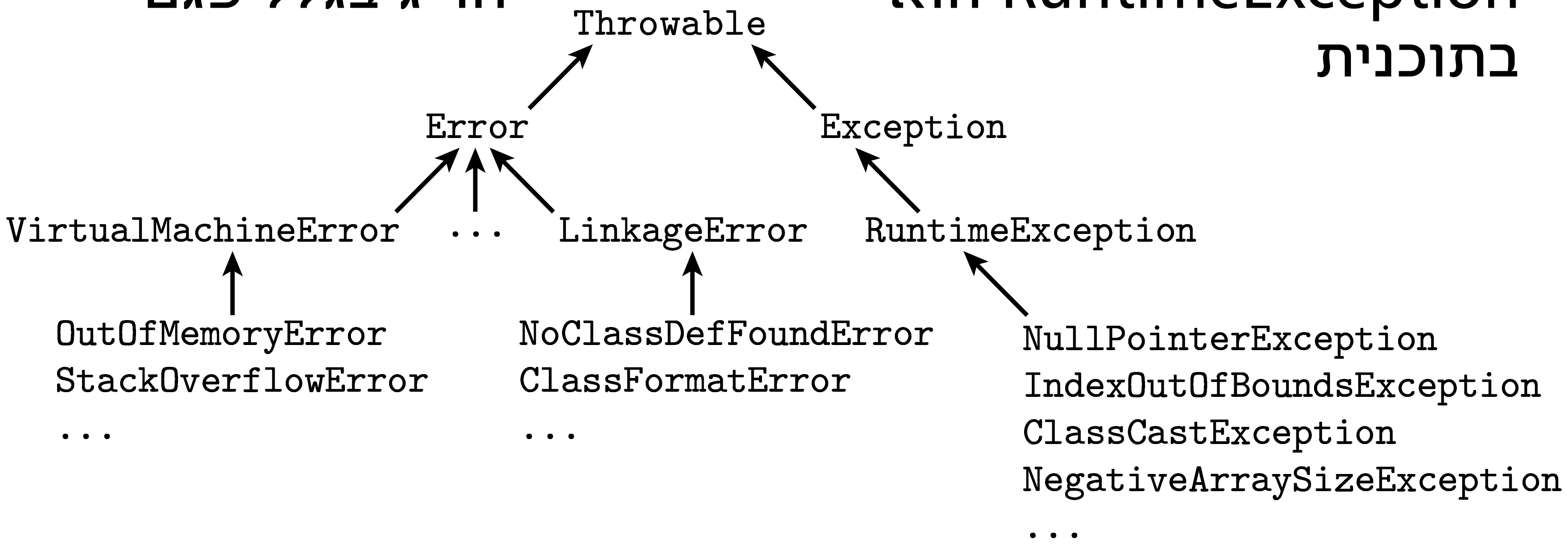
אולי אפשר לנסות שם קובץ אחר, לבקש מהמשתמש להכניס את הדיסקט או התקליטור המתאימים, וכדומה.

טיפוסים חריגים

- בג׳אווה, ההודעה על חריג מתבצעת באמצעות עצם רגיל שמייצג את החריג, את הכישלון של שירות כלשהו
- מכיוון שהחריג הוא עצם רגיל, בונים אותו בעזרת `new`
- הנוהג בג׳אווה הוא לציין את הסיבה לכישלון על ידי טיפוס חריג מתאים, כמו `java.io.FileNotFoundException`
- כמו שראינו בחלק הקודם, אפשר להגדיר בג׳אווה היררכיה שלמה של טיפוסים; ראינו לגבי מנשקים, אבל זה נכון גם לגבי מחלקות
- היררכיה כזו משמשת בג׳אווה לתיאור סיבות כלליות ויותר ספציפיות של חריגים

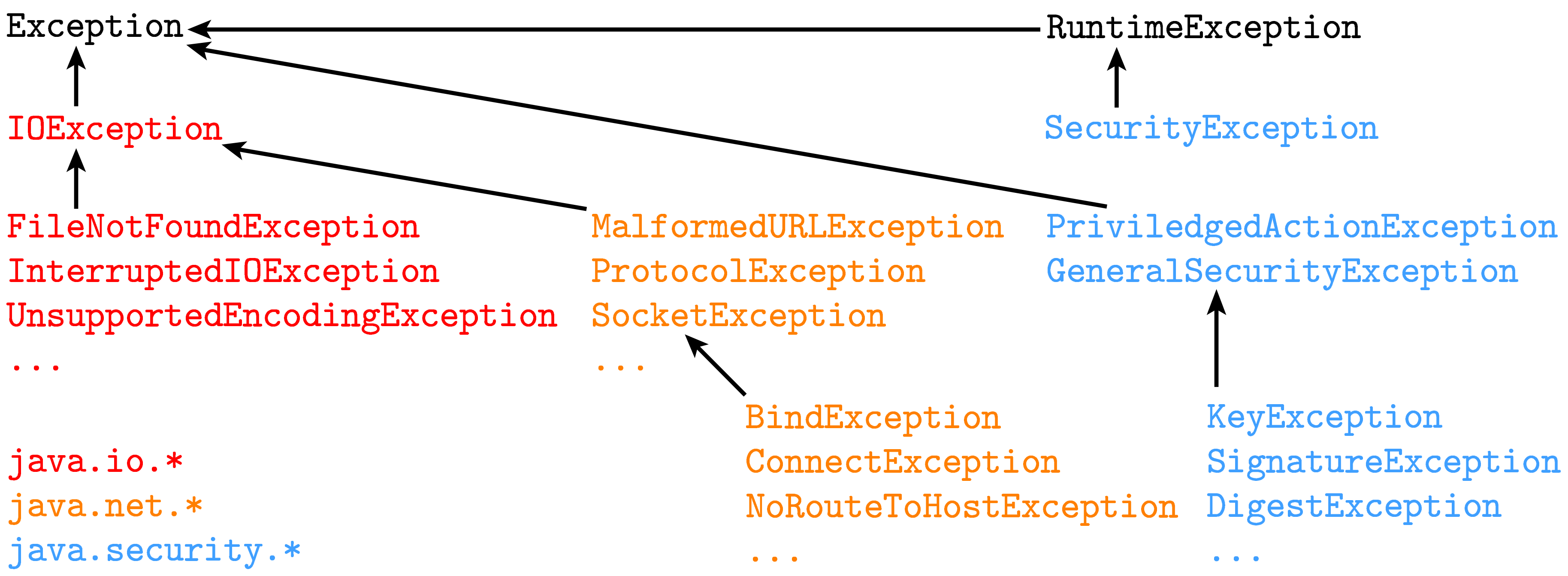
חריגים בחבילה java.lang

- Error: חריגים שמייצגים בעיה בסביבת הריצה, שאינה קשורה בהכרח לתוכנית שרצה: מחסור בזיכרון, קבצי class חסרים או לא תקינים, וכדומה; התגובה הנכונה בדרך כלל היא להפסיק את ריצת התוכנית ולתקן את הסביבה
- Exception: חריג בגלל בעיה בתוכנית; RuntimeException הוא חריג בגלל פגם בתוכנית



חריגים בחבילות אחרות

- רוב החריגים האחרים הם סוגים של Exception ומיעוטם גם של `RuntimeException`
- סוגים של Error מוגבלים לחריגים שסביבת זמן הריצה (ה-virtual machine) מזהה בעצמה



הצהרה על חריגים

```
int compareTo(Comparable other)
   throws IncomparableException {...}
```

- שירות שעשוי להודיע על חריג שאינו סוג של Error או של RuntimeException חייב להצהיר על האפשרות הזו
- לקוח שקורא לשירות שהגדרתו כוללת הצהרה כזו חייב להתייחס לאפשרות שהשירות יכשל ויודיע על חריג
- טיפול אפשרי 1: רק הצהרה שגם הלקוח עשוי להודיע על אותו סוג חריג
- טיפול אפשרי 2: קריאה לשירות בתוך בלוק try עם פסוק catch מתאים (ספציפי או כללי)
- בהיעדר טיפול כזה הלקוח לא יעבור קומפילציה

למה (לא) להצהיר על חריג

- הדרישה להצהיר על חריג מאפשר לקומפיילר לוודא שמי שקורא לשירות מודע לאפשרות של כישלון
- בפרט, זה מונע אפשרות שחריג "יעבור דרך" שירות שלא מתייחס לאפשרות הזו ולכן לא משחזר את המשתמר
- אם זה כל כך מועיל, אז למה יש סוגי חריגים שאין צורך להכריז עליהם?
- מכיוון שחריגים מסוג RuntimeException או Error מוכרזים בגלל פגם בתוכנית או בגלל בעיה לא צפויה במחשב או בסביבת התוכנה שמריצה את התוכנית
- חריגים כאלה אינם צפויים ויכולים לקרות בכל שירות
- בדרך כלל הם גורמים לעצירת התוכנית וכאשר זה המצב, אין חשיבות לשחזור המשתמר

חריגים יוצאים מהכלל

- יוצאים מכלל זה הם ClassCastException, OutOfMemoryError, ואולי עוד כמה חריגים
- בחריג מסוג ClassCastException כדאי לטפל אלא אם בטוחים שיציקה לא אמורה לגרום לו (כלומר אם יתכן חריג שלא בגלל פגם בתוכנית)
- במחסור בזיכרון ניתן לפעמים לטפל: אם קוראים לשירות שזקוק לכמות זיכרון גדולה, כדאי לתפוס את החריג אם השירות מודיע עליו
- במקרה כזה אפשר או לנסות לבצע את הפעולה בדרך אחרת, יותר חסכונית בזיכרון (למשל דרך יותר איטית שמשתמשת בקבצים), או להודיע למשתמש שהפעולה שביקש אינה אפשרית בגלל מחסור בזיכרון

חריגים יוצאים מהכלל

- בשני המקרים אין טעם לסרבל כל קטע קוד ולטפל בחריגים הללו.
- אם למשל התיכון של מחלקה מבטיח שייחוס יהיה מטיפוס מסוים, אין צורך לטפל בחריג ClassCastException למקרה של פגם בתוכנית. אבל אם בתיכון המקורי של התוכנית איננו יודעים מה יהיה הטיפוס של ייחוס, ואנו מבקשים לצקת אותו, אז צריך לטפל בחריג.
- באופן דומה, אין טעם לטפל במחסור בזיכרון אלא במקום שבו יש סיכוי גבוהה למחסור, ושבו יש דרך כלשהי לטפל במחסור. במקרים כאלה, חשוב שכל העצמים ששורדים את הטיפול בחריג ישחזרו את המשתמרים שלהם, וזה יכול לדרוש תפיסה של החריג בכמה רמות של הקוד.

חריג הוא עצם

```
class IncomparableException
  extends Exception {...}
```

- הוא צריך בנאי(ם) ואפשר להוסיף לו שדות מופע ושירותים
- אבל למה עצם?
- באמת לא ברור, הרי הטיפוס של החריג מספיק לסיווגו
- סיבה אפשרית 1: במקרה של חריג בגלל פגם בתוכנית או במערכת המחשב, החזרת מידע שיאפשר לתקן את הפגם
- סיבה אפשרית 2: במקרה של חריג שצריך להודיע עליו למשתמש ("הפעולה נכשלה בגלל ..."), ההודעה למשתמש
- סיבה אפשרית 3: מידע שיאפשר להתאושש (נדיר)
- סיבה לא טובה: בג'אווה כל דבר הוא עצם

חריג כעצם

- בג'אווה לכל החריגים יש לפחות בנאי ריק, בנאי שמקבל מחרוזת, ושירות getMessage שמחזיר את המחרוזת
- מקובל ליצור עצמי חריג עם מחרוזת הסבר, אבל צריך לזכור שמחרוזות כאלה לא מתאימות, בדרך כלל, להצגה למשתמש (המשתמש לא בהכרח דובר אותה שפה של התוכניתן, וכושר הביטוי של תוכניתן לא תמיד מספיק רהוט)
- לא רצוי להגדיר חריגים מורכבים, ובייחוד לא רצוי להגדיר חריגים שהבנאי שלהם עלול להיכשל ולגרום לחריג; זה ימסך את החריג המקורי

שימוש אחר לחריגים

- בשפות שבהן הודעה על חריג ותפיסת חריג זולות, ניתן להשתמש במנגנון החריגים על מנת לממש שירות שיכול להחזיר ערך מאחד מתוך מספר טיפוסים

```
public void polyMethod(...) throws
  resultType1, resultType2 {
  do something
  if (...) throw new resultType1(...);
  else     throw new resultType2(...);
}
```

- לא רצוי בג׳אווה בגלל שמנגנון החריגים יקר מאוד
- חריגים לא נועדו לשמש כעוד מנגנון בקרה

הערות על חריגי חוזה (אין בג׳אווה)

- חריגי RuntimeException מודיעים על פגם בתוכנית
- למשל, NullPointerException מודיע על מעבר על האיסור להשתמש בייחוס שערכו null
- אלה הפרות של חוזה: החוזה של שפת התכנות
- אם החוזה של מחלקה מפורש ומוגדר בעזרת פסוקים בוליאניים חוקיים בג׳אווה, אפשר אולי לבדוק עמידה בחוזה
- כאשר מתגלה אי עמידה בחוזה בכניסה או ביציאה משירות, מודיעים על חריג מסוג אי עמידה בתנאי קדם, תנאי אחר, או אי קיום המשתמר
- בג׳אווה אין תמיכה מובנית בחריגי חוזה; בשפות אחרות יש

ג׳אווה וחריגי חוזה

גישה פשוטה:

```
public String getVersion(int i) {
  if (i > length() || i <= 0)
    throw new PreconditionException();
```

- שלוש בעיות: ראשית, החוזה והקוד מתערבבים; בעייתי בייחוד אם יש כמה נקודות חזרה (פסוקי return או throw)
- שנית, לא תואם למנגנון חשוב בתכנות מונחה עצמים שעוד לא נגענו בו, ירושה
- שלישית, אין דרך פשוטה שלא לבדוק את החוזה אם הבדיקה יקרה; צריך דרך כזו ברמת התוכנית ו/או ברמת המחלקה
- ישנם תוספים חיצוניים שמספקים תמיכה בחוזים.

כלים לתמיכה בחוזים וחריגי חוזה בג'אווה

- הכלים שידועים לנו, חלקם חינם וחלקם מסחריים הם: JMSAssert, iContract, jContractor, Handshake, JML, Jass, JPP, Jose

חריגים גרועים

- מפתחים משתמשים בחריגים לעוד מטרות, פחות מוצדקות
- השימוש הגרוע ביותר הוא על מנת לחסוך שאילתה זולה
- דוגמה: ספריית הקלט/פלט של ג'אווה תומכת במספר קידודים (encodings) עבור קבצי טקסט, אבל לגרסאות שונות של הספרייה מותר לתמוך במבחר קידודים שונה
- אין דרך לשאול האם קידוד נתמך או לא
- אבל אם מנסים להשתמש בקידוד לא נתמך, השירות מודיע על חריג java.io.UnsupportedEncodingException
- עדיף היה לברר האם קידוד נתמך בעזרת שאילתה
- שאלה: האם EOFException מוצדק? (סוף קובץ). אולי עדיפה שאילתה?

חריג או שאילתה?

```
class TaggedVersionedString {
  ... //  some implementation of VersionedString

  public void   tag(int i, String t) {...}
  requires: t is not an existing tag in this object
            & 1<=i<=length()
  ensures: getVersion(t)==getVersion(i)
  public String getVersion(String t) {...}
  requires: t is an existing tag in this object
  ensures: return_value != null
}
```

שני גישות לתנאי הקדם

- כרגע תנאי הקדם של tag לא נוח, כי אין דרך לבדוק האם מחרוזת נתונה כבר משויכת לגרסה
- אפשרות אחת היא להוסיף שאילתה שתענה על השאלה, ואז לקוח טיפוסי יראה כך:

```
if ( !vs.exists(t) ) vs.tag(i,t);
else ...  tell the user that  she can't use the tag t
```

- או שאפשר לבדוק את זה בשירות tag ולהודיע על חריג אם המחרוזת כבר משויכת לגרסה, ואז לקוח טיפוסי יראה כך:

```
try { vs.tag(i,t); }
catch (DuplicateTagException e)
  { ...  tell the user that  she can't use the tag t  }
```

לכאורה אין הבדל גדול

- והשימוש בחריג נראה יותר "חסין", מכיוון שהוא דורש פחות מהלקוח והספק מבטיח יותר (בפרט מבטיח לבדוק תקינות)
- אבל בשימושים אחרים במחלקה, יותר מורכבים, יש הבדל לטובת השימוש בשאילתה

אבל לפעמים יש הבדל: זה עובד

```
static void tagAll(VersionedString[] a,
                   String t) {
  boolean duplicate = false;
  int i;
  for (i=0; i<a.length; i++)
    if ( a[i].exists(t) ) duplicate = true;
  if (!duplicate)
    for (i=0; i<a.length; i++)
      a[i].tag(a[i].length(),t);
  else ...  tell the user that she can't use the tag t
}
```

אבל זה לא...

```
static void tagAll(VersionedString[] a,
                   String t) {
  boolean duplicate = false;
  int i;
  try {
    for (i=0; i<a.length; i++)
      a[i].tag(a[i].length(),t);
  } catch (DuplicateTagException e) {
    now what? some elements have already been tagged,
    and we don't have a method  to remove tags!
}
```

אולי גם שאילתה וגם חריג?

- אפשרי, אבל לא יעיל ומעיק
- גם כאשר הלקוח יודע בוודאות שהספק יכול לבצע את השירות (למשל, המחרוזת לא משויכת לאף גרסה), הוא חייב לעטוף את הקריאה לספק בפסוק try-catch
- בנוסף לסרבול, הספק תמיד בודק תקינות, גם כאשר בוודאות אין בכך צורך
- שימוש בשאילה מסורבל בערך כמו חריג, מאפשר להימנע מהבדיקה כשלא צריך אותה, ומונע את הצורך לנסות לבצע את הפעולה על מנת לדעת אם תצליח

עצם סטאטוס במקום חריג

- אפשר להחליף חריג בעצם סטאטוס שדרכו הספק ידווח על כישלון, למשל

```
Matrix a = ...;
Vector b = ...;
Vector x;
SolveStatus s = new SolveStatus();
x = a.solve(b, s);
if (s.succeeded())              {...}
else if (s.failedSingular()) {...}
```

- פחות יעיל במקרה של הצלחה; יותר יעיל במקרה כשלון

פקודה ושתי שאילתות במקום חריג

- אפשר להחליף חריג בשתי שאילתות, לבדוק אם הפעולה הצליחה, ואם כן לקבל את התוצאה, למשל

```
Matrix a = ...;
Vector b = ...;
Vector x;
a.try_to_solve();
if (a.succeeded())
   b = s.solution()
else ...
```

גישה לטיפול במקרים לא נורמליים

- שלוש גישות לטיפול במקרים בהם ההתנהגות שונה מהרגיל, כאשר לקוח מבקש שרות מספק, ולא ניתן לספקו:
- טיפול א-פריורי: הלקוח בודק בעזרת שאילתת ספק את תנאי הקדם (או שאינו בודק, אם בטוח שהתנאי חייב להתקיים). אם התנאי לא מתקיים, הלקוח לא מבקש שרות.
- טיפול א-פוסטריורי: אם בדיקת התנאי יקרה או בלתי מעשית, הלקוח מבקש מהספק לנסות לתת את השרות, ומברר אם השרות הסתיים בהצלחה, בעזרת שרותי הספק.
- שימוש בחריגים אם שתי הגישות האלה לא מתאימות (למשל אם ארוע לא רגיל גורם לחריג חומרה או מערכת הפעלה).

סיכום חריגים

- חריגים מודיעים על כשלון של ספק לקיים את תנאי האחר, למרות שהלקוח קיים את תנאי הקדם
- חריג הוא מוצדק כאשר אי אפשר לדרוש מהלקוח לקיים תנאי קדם שיבטיח את הצלחת השירות, או כי ללקוח אין מספיק שליטה, או כי בדיקה על ידי הלקוח יקרה מדי, או כי קשה להגדיר תנאי קדם תמציתי
- חריג אינו מוצדק אם הלקוח היה יכול למנוע אותו בעזרת שאילתה פשוטה
- חריג הוא עצם לכל דבר אבל עדיף להשתמש בו כאיתות ותו לא
- טיפול בחריג בלקוח: שחזור המשתמר והודעה ללקוח שלו על חריג (אולי אחר) או ביצוע המשימה שלו בדרך אחרת