תוכנה 1 בשפת Java
שיעור מספר 9: "יוצא מן הכלל"

אוהד ברזילי
דן הלפרין

בית הספר למדעי המחשב
אוניברסיטת תל אביב

היום בשיעור

- **חריגים (Exceptions)** מבטאים מצבים יוצאי דופן, מקרי קצה ומצבים בלתי צפויים בריצת התוכנית כגון: ארגומנטים שאינם חוקיים, בעיות ברשת התקשורת, קובץ שאינו קיים ועוד
- על הקשר בין חריגים לחוזים וליחסי ספק-לקוח
- נדון בגישות שונות להתמודדות עם מקרי קצה ונעמוד על היתרונות והחסרונות שלהן (כרגיל, אין פתרונות קסם)

אם יש ישאר זמן -

- **טענות (Assertions)** – מבטאות הנחות שיש למתכנת בנקודה מסויימת בקוד
  - בזמן ריצה, ניתן להסיר את הטענות מן הקוד לחלוטין ובכך לא להאט את ריצת התוכנית

לא כל החריגים אותו הדבר

- תנאים אשר עשויים להתקיים במהלך ריצה תקין של תוכנית תקינה (מקרי קצה) נקראים ***checked exceptions***
  - תנאים אלו מיוצגים ע"י המחלקה **`Exception`**

- בעיות חמורות הנחשבות קטלניות (fatal), וכן מצבים המייצגים שגיאות בתוכנית (bugs) נקראים ***unchecked exceptions***
  - בעיות חמורות מיוצגות ע"י המחלקה **`Error`**
  - שגיאות בתוכנית מיוצגות ע"י המחלקה **`RuntimeException`**

- תיעוד המחלקות האוטומטי (javadoc API) מתאר עבור כל מתודה את ה- *checked exceptions* שהיא עשויה לחולל (ועשוי לתאר גם *unchecked exceptions*)

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
```

- איך נוודא כי הקובץ קיים?
- גם 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

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

```java
public class AddArguments {
   public static void main(String args[]) {
      int sum = 0;
      for (String arg : args) {
        sum += Integer.parseInt(arg);
      }
      System.out.println("Sum = " + sum);
   }
}
```

"הגורם האנושי"

> java AddArguments 1 2 3 4
Sum = 10

> java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:447)
at java.lang.Integer.parseInt(Integer.java:497)
at AddArguments.main(AddArguments.java:5)

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s` - a `String` containing the `int` representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` - if the string does not contain a parsable integer.

- לשרות אין תנאי קדם (true), ואולם הוא בחר לטפל בקלטים מסוימים, **שלא ע"י החזרת ערך**, אלא ע"י **זריקת חריג**
- זוהי הגדרת **תנאי צד** (side condition) – **הלקוח** אינו מחויב לקיים את תנאי הצד לפני הקריאה לשרות. תנאי הצד משמש "נתיב מילוט" **לספק**
- שימו לב, הדבר שונה מהגדרת תנאי קדם משמעותי, שבו השרות **מניח** שתנאי הקדם מתקיים, **ומתעלם** ממקרים שבהם הוא אינו מתקיים

תנאי קדם והספרייה התקנית

- בספריות התקניות של שפת Java מקובל להגדיר שרותים **ללא תנאי קדם** (true), אך עם **תנאי צד** משמעותיים אשר מוגדרים להם חריגים מתאימים

- כבר ראינו כי לספקים סובלניים יש חסרונות, אולם השימוש בחריגים מנסה לפתור את חלקם

- במהלך השיעור ננסה לעמוד על היתרונות והחסרונות של הגישות השונות

try-catch block

```
public class AddArguments {
    public static void main(String args[]) {
        try {
            int sum = 0;
            for ( String arg : args ) {
                sum += Integer.parseInt(arg);
            }
            System.out.println("Sum = " + sum);
        } catch (NumberFormatException nfe) {
            System.err.println("One of the command-line "
                    + "arguments is not an integer.");
        }
    }
}
```

```
> java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```

טיפול בחריגים בג'אווה

- חריג יכול להיזרק ע"י פקודת **`throw`** (נראה בהמשך).
- קטע קוד אשר עלול לזרוק חריג יעטף ע"י הלקוח **בבלוק `try`**
- פקודת **`throw`** גורמת להפסקת הביצוע הרגיל, והמשערך מחפש exception handler (**בלוק catch**) שיתפוס את החריג.
  - אם בלוק ה catch 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.

גרעיניות

```
public class AddArguments3 {
    public static void main(String args[]) {
        int sum = 0;
        for ( String arg : args ) {
            try {
                sum += Integer.parseInt(arg);
            } catch (NumberFormatException nfe) {
                System.err.println("[" + arg + "] is not an integer"
                    + " and will not be included in the sum.");
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

```
> java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the sum.
[3.0] is not an integer and will not be included in the sum.
Sum = 5
```

ריבוי בלוקי catch

- לבלוק try אחד עשויים להיות כמה בלוקים של catch השייכים לו, עבור סוגים שונים של שגיאות שעשויות לקרות:

```
try {
    // code that might throw one or more exceptions
} catch (MyException e1) {
    // code to execute if a MyException exception is thrown
} catch (MyOtherException e1) {
    // code to execute if a MyOtherException exception is thrown
} catch (Exception e3) {
    // code to execute if any other exception is thrown
}
```

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

בלוק finally

- קטע קוד המופיע בבלוק **finally** יתבצע בכל מקרה (בין אם קטע הקוד בבלוק ה **try** הצליח או נכשל)

```
try {
   startFaucet();
   waterLawn();
} catch (BrokenPipeException e) {
   logProblem(e);
} finally {
   stopFaucet();
}
```

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 לתוכניתנית 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) {
  IPoint other_point;
  other_point = (IPoint) other;
  if (this.x() > other_point.x())
      ...
```

- המרת טיפוסים (שנדון בה בשיעור הבא) עלולה להודיע על חריג אם העצם (other) אינו מטיפוס שמתאים לניסיון ההמרה (כאן **IPoint**)
- 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) {
 IPoint other_point;
  try {
     other_point = (IPoint)other;
  }
  catch (java.lang.ClassCastException ce) {
    throw new IncomparableException();
  }
  if (this.x() > other_point.x())
    …
```

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

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 למָסֶך 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

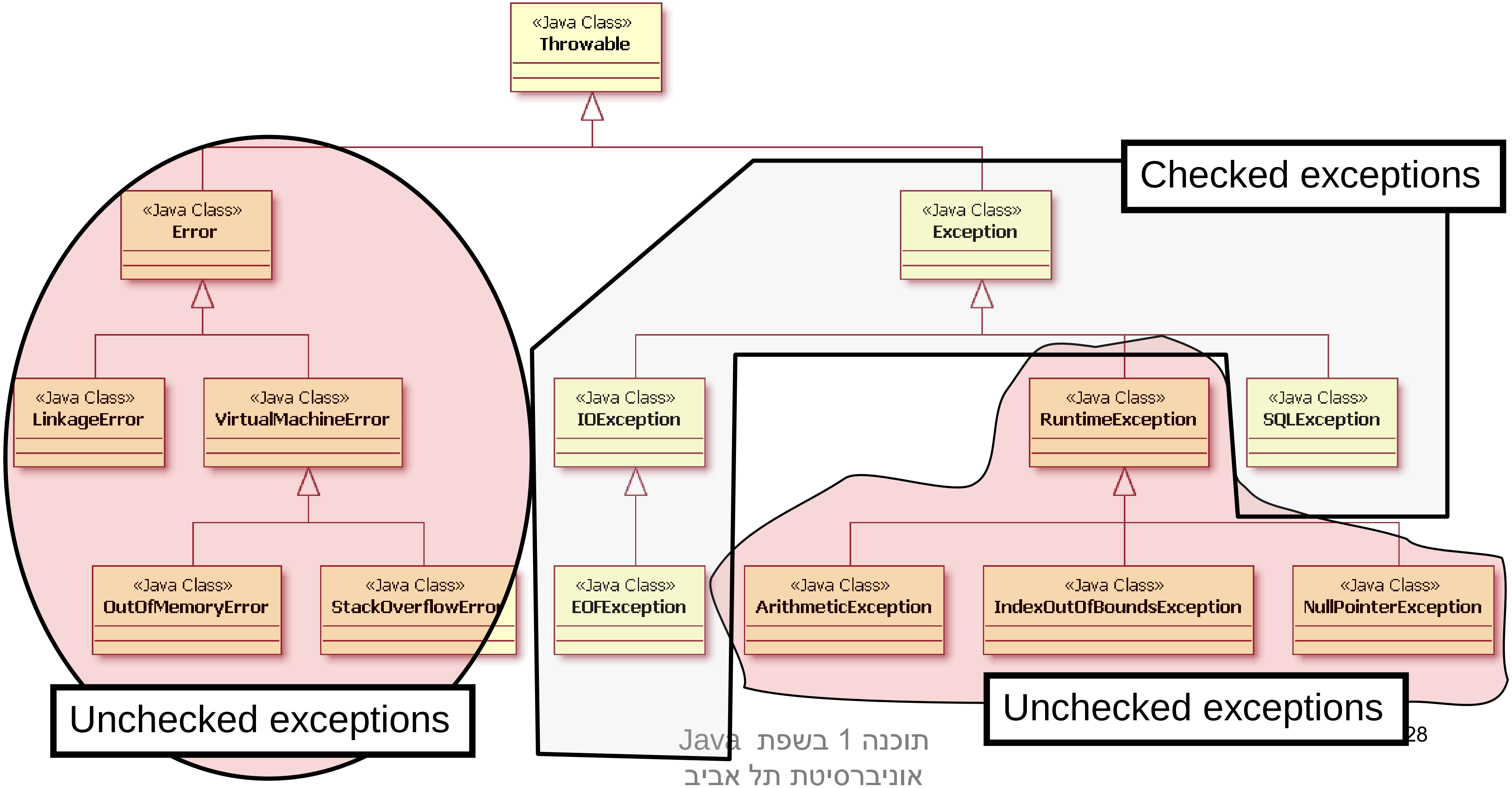
- בג'אווה, ההודעה על חריג מתבצעת באמצעות עצם רגיל שמייצג את החריג, את הכישלון של שירות כלשהו
- מכיוון שהחריג הוא עצם רגיל, בונים אותו בעזרת **new**
- הנוהג בג'אווה הוא לציין את הסיבה שגרמה לכישלון על ידי טיפוס חריג כמו **java.io.FileNotFoundException**
- ג'אווה מגדירה היררכיה של טיפוסים (מחלקות) עבור חריגים עפ"י הסיבה. המחלקה הכללית ביותר היא **Throwable**, אך החלוקה העיקרית היא לשלוש משפחות:
  - **Error**
  - **RuntimeException**
  - **Exception** שאינו **RuntimeException**

חריגים בחבילה java.lang

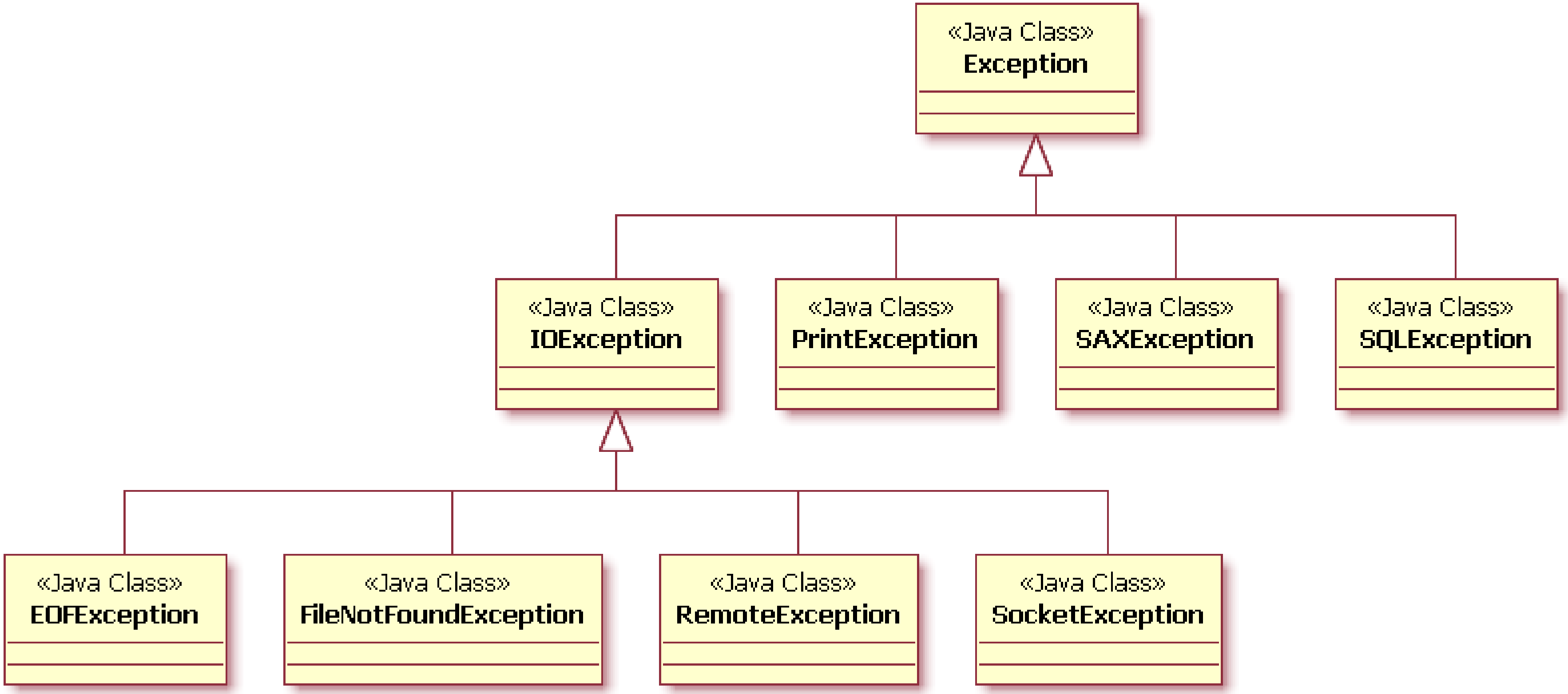
- **Error**: חריגים שמייצגים בעיה שלא ניתן בדרך כלל להתאושש ממנה:
  - בדרך כלל בעיה בסביבת הריצה: מחסור בזיכרון, קבצי class חסרים או לא תקינים, וכדומה; התגובה הנכונה בדרך כלל היא להפסיק את ריצת התוכנית ולתקן את הסביבה.
  - אולם, ניתן להגדיר חריגים מטיפוס Error כדי לבטא שבירה של הנחה לוגית (לדוגמא: AssertionError)

- **Exception**: מתחלקים לשתי קבוצות:
  - **RuntimeException** הוא חריג שיכול לקרות כמעט בכל שירות: גישה למצביע null, כשלון בהמרה, חריגה מתחום מערך וכו'
    - לא אמור להופיע בתוכנית תקינה
  - **Exception** שאינו **RuntimeException** מתרחש במצבים מוגדרים היטב, שניתן לתכנן מראש לקראתם.

היררכיית שגיאות וחריגים (חלקית)



checked exceptions (רשימה חלקית)



הצהירו או טפלו

- קוד המכיל קריאה למתודה שעשויה לחולל (*לזרוק*) **חריג נבדק** (checked) צריך לנקוט אחת משתי הגישות:
  - **הכרזה** על החריג הפוטנציאלי

  או
  - **טיפול** בו

- **טיפול** ראינו, נעטוף את הבלוק הבעייתי בבלוק **`try-catch-finally`**

- **הכרזה** – נשתמש במילה השמורה throws כדי לציין את המתודה העוטפת כולה כ"בעייתית":

```
void trouble() throws IOException { ... }
void trouble() throws IOException, MyException { ... }
```

הצהירי או טפלי

- השימוש במתודות הזורקות חריגים **מידבק** (רקורסיבית)
- אי הכרזה או טיפול גורר שגיאת קומפילציה

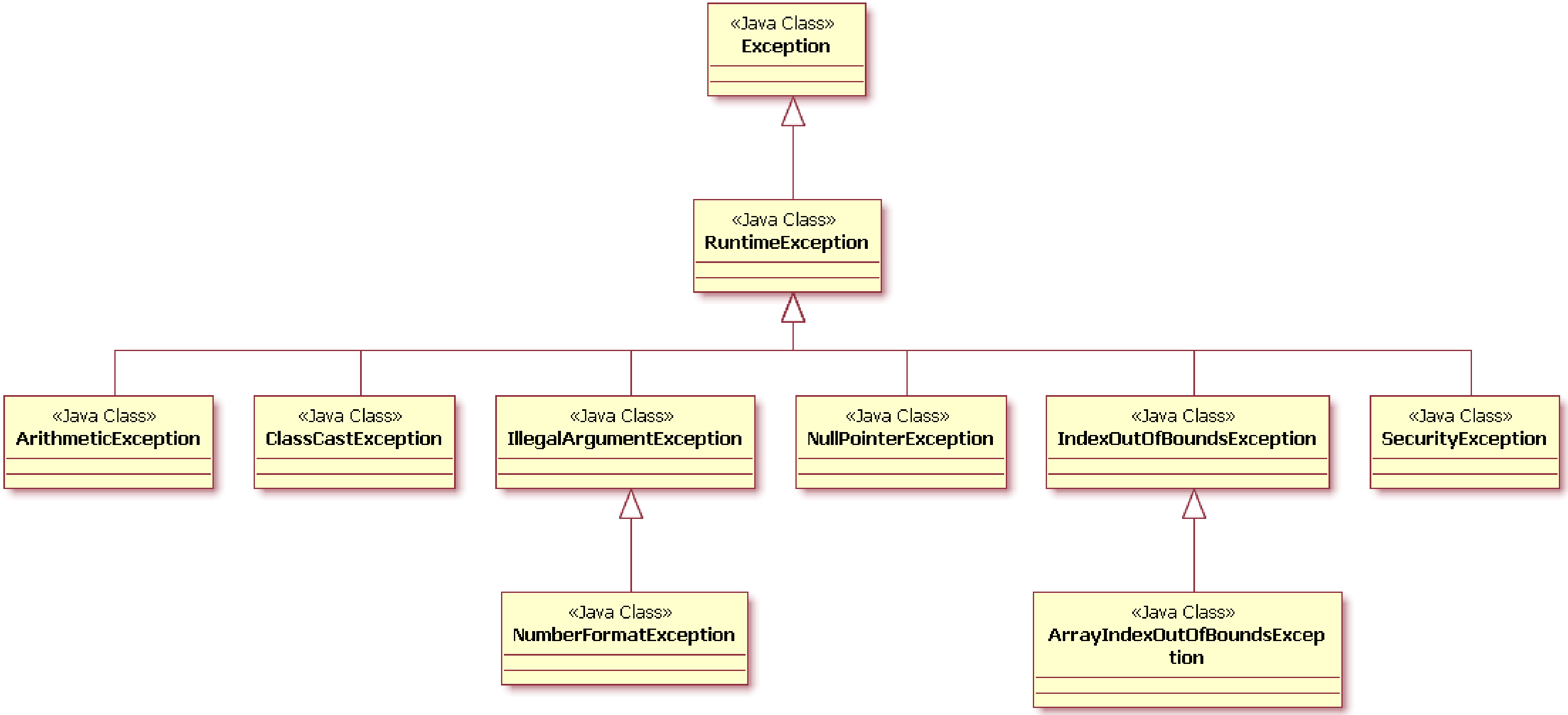
```
void f(int x) throws Exception {...}


void g() {
    f(1); // error
}
```

Declaration

Compilation Error: programmer must either catch the exception, or declare that "void *g()* throws Exception"

(רשימה חלקית) *unchecked exceptions*



unchecked exceptions

- על חריגים או שגיאות שהם unchecked אין חובה להצהיר בחתימת המתודה
  - אבל מומלץ להצהיר עליהם

- בחריגים או בשגיאות שהם unchecked אין חובה לטפל בבלוק try-catch-finally
  - ניתן לחשוב על טיפול בררני

- מדוע?

למה יש חריגים
שלא **חייבים** להצהיר עליהם?

- 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 מועיל, למה יש חריגים שלא **חייבים** להכריז עליהם?

- מכיוון שחריגים מסוג `RuntimeException` או `Error` מוכרזים בגלל פגם בתוכנית או בגלל **בעיה לא צפויה במחשב** או בסביבת התוכנה שמריצה את התוכנית או **באג בתוכנית** שהיה אמור להתגלות בתהליך הפיתוח
  - חריגים כאלה אינם צפויים ויכולים לקרות בכל שירות
  - בדרך כלל הם גורמים לעצירת התוכנית וכאשר זה המצב, אין חשיבות לשחזור המשתמר

- דיון: איך נערך ל:
  - OutOfMemoryError
  - ClassCastException
  - FileNotFoundException

הגדרת חריגי משתמש

- מנגנון זריקת ותפיסת החריגים הינו חלק משפת התכנות, אולם החריגים עצמם הם עצמים "רגילים"
- פרט למספר קטן של חריגים שנזרקים ע"י ה JVM רוב החריגים נזרקים כתוצאה מבדיקות שנעשו בקוד "רגיל"
- אנו כמתכנתים יכולים (ולפעמים נדרשים) להגדיר חריגים חדשים ע"י הגדרת מחלקה חדשה היורשת מהמחלקה **`Throwable`** או אחת מצאצאיה
- בחירת ההורה תלויה בסוג השגיאה שברצוננו להגדיר ובמידה שבה אנו מעוניינים להגביל את לקוחותינו
  - **`RuntimeException`** או **`Error`** – מאפשר ללקוחות מסוימים להתעלם מהאפשרות לחריג
  - **`Exception`** – מחייב את כל הלקוחות להצהיר או לטפל

חריג הוא עצם

```
class IncomparableException extends Exception {…}
class OverdraftException extends RuntimeException {…}
```

- הוא צריך בנאי(ם) ואפשר להוסיף לו שדות מופע ושירותים
- אבל למה עצם?
- באמת לא ברור, הרי הטיפוס של החריג מספיק לסיווג
  - **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

הדפסת מחסנית הקריאות

- אחת המתודות השימושיות של המחלקה **Throwable** היא המתודה **printStackTrace** המדפיסה את שרשרת הקריאות שהובילה לחריג
- זהו גם מימוש ברירת המחדל של ה- JVM לחריג שלא טופל

```
public static void main(String[] args){
    try {
        riskyMethod();
    }
    catch(Exception ex){
       ex.printStackTrace();
    }
    System.out.println("Continuing main …");
}
```

זריקת חריג ע"י הספק

- במחלקה **BankAccount** שראינו בתרגול, כתבנו את המתודה **withdraw** אשר תנאי הקדם שלה היה שסכום המשיכה תקין
- ניתן לחשוב על מתודה **withdraw** ללא תנאי קדם שתזרוק חריג במקרה של נסיון למשיכת יתר

```
class OverdraftException extends RuntimeException{}

public class BankAccount {
    //...

    public void withdraw(double amount)
                                    throws OverdraftException {
      if (amount < balance)
       throw new OverdraftException();
      balance -= amount;
    }
```

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

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?

תנאי צד או תנאי קדם?

- כאשר הפכנו את **תנאי הקדם** של **`withdraw`** להיות **תנאי צד**, השפענו על הצורה שבה משתמשים בשרות
- במקום שלקוח טיפוסי יראה כך:

```
if (acc.balance() >= amount)
   acc.withdraw(amount);
else
   // tell the user that she can't withdraw
```

- השירות **`withdraw`** בודק את התנאי בעצמו ומודיע על חריג אם הפעולה אסורה, ואז לקוח טיפוסי יראה כך:

```
try {
    acc.withdraw(amount);
} catch (OverdraftException e) {
   // tell the user that she can't withdraw
}
```

לכאורה אין הבדל גדול

- השימוש 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

אולי גם שאילתה וגם חריג?

- אפשרי, אבל לא יעיל ומעיק
- גם כאשר הלקוח יודע בוודאות שהספק יכול לבצע את השירות (בדק את היתרה בעצמו), הוא חייב לעטוף את הקריאה לספק בפסוק try-catch
  - ניתן להימנע מכך ע"י הגדרת החריג כ **`RuntimeException`**
- בנוסף לסרבול, הספק תמיד בודק תקינות, גם כאשר בוודאות אין בכך צורך
- שימוש בשאילתה 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.closeToSingular ()) {…}
```

- 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(b);
if (a.succeeded())
   x = a.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 או אם לספק תנאי צד מסובך - 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).

ירושה וחריגים

- בג'אווה פסוק **throws** (ליתר דיוק להעדרו) הוא חלק מחוזה.

- שרות שמממש שרות מופשט (ממחלקה מופשטת שירש, או ממנשק שהוא ממשש) או שדורס שרות שירש, **רשאי** לכלול פסוק **throws** עבור חריג נבדק E , רק אם השרות אותו הוא יורש כולל פסוק כזה עבור E או עבור מחלקה כללית יותר מ E. אחרת הקומפיילר יוציא הודעת שגיאה.

- אבל **מותר** לשרות היורש לא לכלול פסוק **throws** עבור חריג נבדק E שהיה פסוק כזה עבורו בשרות המוריש. במקרה זה החוזה במחלקה היורשת חזק יותר: היא מבטיחה שהשרות לא יזרוק את E , למרות שהחוזה שירשה מרשה זאת

- כשמתכננים מנשק יש לכלול פסוקי **throws** לפי הצורך

כלל אצבע

למתודה דורסת (או מממשת) מותר לזרוק:

- אף חריג
- חריגים שזרקה המתודה הנדרסת
- חריגים היורשים מחריגים שזרקה המתודה הנדרסת

למתודה דורסת (או מממשת) אסור לזרוק:

- חריגים שלא זרקה המתודה הנדרסת
- חריגים המהווים מחלקות בסיס לחריגים שזרקה המתודה הנדרסת

חריגים וירושה - דוגמא

```
public class TestA {
   public void methodA() throws IOException {
        // do some file manipulation
   }
}

public class TestB1 extends TestA {
   public void methodA() throws EOFException {
        // do some file manipulation
   }
}

public class TestB2 extends TestA {
   public void methodA() throws Exception {
        // do some file manipulation
   }
}
```

ריבוי בלוקי catch וירושה

- אם יש כמה פסוקי catch מתאימים יתבצע הבלוק המתאים הראשון (לא בהכרח המתאים ביותר!)

```
void readData() throws SQLException, IOException {
    …
}

void test(){
    try {
        readData();
    }
    catch(SQLException e){ … }
    catch(IOException e) { … }
}
```

ריבוי בלוקי catch וירושה

```
void readData() throws SQLException, EOFException {
    …
}

void test(){
    try {
        readData();
        Thread.sleep(100);
    }
    catch(SQLException e){ … }
    catch(IOException e) { … }
    catch(Throwable e)   { … }
}
```

טענות (assertions)

- תחביר:

assert <boolean_expression> ;

assert <boolean_expression> : <detail_expression> ;

- אם הביטוי boolean_expression משתערך ל **`false`** התוכנית זורקת **`AssertionError`**

- הביטוי detail_expression הופך למחרוזת לתיאור מהות השגיאה

דוגמאות שימוש

- טענות מבטאות הנחות שיש למתכנת על הלוגיקה הפנימית בקטע קוד מסוים

- לדוגמא:
  - שמורה פנימית
  - שמורת מבני בקרה (control flow invariant)
  - שמורת מחלקה ותנאי בתר

שמורה פנימית

```
if (x > 0) {
   // do this
} else {
   // do that
}
```

אבל אם ידוע ש x אינו יכול להיות שלילי, עדיף:

```
if (x > 0) {
   // do this
} else {
   assert ( x == 0 );
   // do that, unless x is negative
}
```

שמורת מבני בקרה

```
switch (suit) {
   case Suit.CLUBS: // ...
        break;
   case Suit.DIAMONDS: // ...
        break;
   case Suit.HEARTS: // ...
        break;
   case Suit.SPADES: // ...
        break;
   default: assert false : "Unknown playing card suit";
        break;
}
```

שימוש ב assert לחוזים

- ניתן לבצע מעקב אחרי חוזים ע"י כתיבת שרות מיוצא כך:

```
public ... method(... ) {
   assert(pre1) : "pre1 in words";
   assert(pre2) : "pre2 in words";
   assert(inv1) : "inv1 in words";
   ... // the body of method
   assert(post1) : "post1 in words";
   assert(post2) : "post2 in words";
   assert(inv1) :  "inv1 in words";
}
```

שימוש ב assert לחוזים (המשך)

- .. pre1, pre2, הם פסוקים שונים של תנאי הקדם.
- .. post1, post2, הם פסוקים שונים של תנאי האחר.
- .. inv1, inv2, הם פסוקים שונים של המשתמר.

- זה אינו פתרון מספק:
  - המתכנת צריך לטפל בעצמו ב $prev ,
  - לכתוב את המשתמר פעמיים בכל שרות, ...
  - תנאי קדם של בנאי צריך להיבדק לפני הכניסה לבנאי

- זה אינו פתרון אידיאלי. עדיף כלי שנועד לחוזים. אבל אם אין ברשותנו כלי, ניתן להשתמש חלקית במשפטי assert

כלים לתמיכה בחוזים

- כלי שמתרגם את החוזה שכתבנו בתוך הערות ה doc ויוצר עבורנו משפטי assert (או משפטים דומים).
- הכלי צריך גם לקחת חוזה ממנשק או מחלקה ממנה ירשנו ולהוסיף את החלק המחזק/מחליש
- לחליפין, הכלי יבדוק שהחוזה במחלקה היורשת מחזק את תנאי הבתר ומחליש את תנאי הקדם (בשיעור הבא)
- רצוי שהכלי יציג את החוזה ויבחין בעצמו בין החלק המיוצא של החוזה לחלק החסוי.
- הכלים שידועים לנו, חלקם חינם וחלקם מסחריים הם: JMSAssert, iContract, jContractor, Handshake, JML, Jass, JPP, Jose

דוגמא נאיבית

- כלי אכיפה נאיבי יבצע גזירה (parsing) של משפטי החוזה מתוך הטענות, והדבקה שלהם (instrumentation) במקום המתאים

- לדוגמא, הקוד הבא:

```
/** @post getValue() == newValue, “value is updated”*/
public void setValue(int newValue) {
    val = newValue;
}
```

- יהפוך ל:

```
public void setValue(int newValue) {
    val = newValue;
    assert(getValue() == newValue : “value is updated”);
}
```

טענות וביצועים

- כברירת מחדל אכיפת הטענות אינה מאופשרת
- יש לאפשר זאת במפורש:

```
> java -enableassertions MyProgram
```

או

```
> java -ea MyProgram
```

- ניתן לשלוט באכיפת טענות עבור מחלקה מסויימת, חבילה או היררכיית חבילות. הפרטים המלאים:

http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html

סיכום חריגים

- חריגים מודיעים על כשלון של ספק לקיים את תנאי האחר, למרות שהלקוח קיים את תנאי הקדם
- חריג הוא מוצדק כאשר לא ניתן 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 רק כאיתות
- טיפול בחריג בלקוח: שחזור המשתמר והודעה ללקוח שלו על חריג (אולי אחר) או ביצוע המשימה שלו בדרך אחרת