חלק 7
חריגים

לפני כן - חזרה למחלקות גנריות:

מה עושים ללא מחלקות גנריות

- רוצים לכתוב מנשק ו/או מחלקה עבור מחסנית, שתאפשר ליצור מחסנית של שלמים, ומחסנית של מחרוזות, וכו'
- אם אין אפשרות להשתמש במנשק/מחלקה גנרית (למשל בג'אווה 1.4) נצטרך להניח שהאברים הם מהמחלקה הכללית ביותר, כלומר Object
- בדוגמא - מנשק למחסנית, ומחלקה ממשת (ללא החוזה).

```
interface Stack {
  public Object top ();
  public void push(Object t);
  public void pop();
  public boolean empty();
  public boolean full();
}
```

```java
public class FixedCapacityStack
        implements Stack{
    private Object [] content;
    private int capacity;
    private int topIndex;
    public FixedCapacityStack(int capacity){
        content = new Object[capacity];
        this.capacity = capacity;
        topIndex = -1;
    }
    public Object top () {
        return content[topIndex];   }
```

```
  public void push(Object t) {
     content[++topIndex] = t;    }
  public void pop() {
     topIndex--; }

  public boolean empty() {
     return (topIndex < 0);
  }

  public boolean full() {
     return (topIndex >= capacity - 1) ;
  }
}
```

איך נשתמש במחסנית?

- נניח שרוצים מחסנית של מחרוזות

```
Stack s = new FixedCapacityStack(5);
s.push("hello");
String t1 = s.top(); // compilation error
String t2 = (String) s.top(); //ok
```

- באחריות המתכנת לוודא שכל האברים המוכנסים למחסנית הם מאותו טיפוס (כאן מחרוזות), אחרת היציקה תיכשל.
- בדיקת היציקה נעשית בזמן ריצה. אנחנו מאבדים בטיחות טיפוסים.
- פתרון אחר: מנשק/מחלקה נפרדת לכל טיפוס איבר - שכפול קוד!

חזרה למחלקה מוכללת (גנרית)

- מנגנון ההכללה מיועד לאפשר שימוש חוזר בקוד בלי לאבד מידע לגבי הטיפוס הסטאטי של עצם
- בלי הכללה, שימוש חוזר בקוד מתבצע על ידי השמת התייחסות מטיפוס אחד לטיפוס אחר, יותר כללי; מאותו רגע אין דרך לשחזר את הטיפוס הסטאטי המקורי בלי יציקה
- תפקיד ההכללה הוא למנוע צורך ביציקות, שנבדקות מאוחר
- הפרטים מסתבכים בגלל האינטראקציה בין מנגנון ההכללה ובין יחס הירושה (יחס ה-is-a)
- קושי נוסף: תאימות בין גרסאות גנריות ולא גנריות

איך זה עובד

- הקומפיילר ממפה את כל המחלקות המוכללות FCStack<Something> למחלקה אחת רגילה (לא מוכללת) שהיא בעצם FCStack<Object>
- בקוד שמשתמש במחלקה מוכללת, הקומפיילר מוסיף לקוד יציקות על מנת לבצע השמות מ-Object לטיפוס הספיציפי, למשל String
- הקומפיילר מוודא שהיציקה תמיד תצליח ולעולם לא תודיע על ClassCastError,

```
String t = (String) s.top();
```

- כלומר, הטיפוס המוכלל (T) נמחק מהקוד שהקומפיילר מייצר; הוא שימושי רק לבדיקות תקינות טיפוסים בזמן קומפילצייה; התהליך נקרא מחיקה (erasure)

הכללה ויחס is-a

```
Stack <String>  ts =
    new FCStack <String> (5);
Stack <Object> to =
    new FCStack <Object> (5);
```

`to = ts;`	*should this work?*
`ts.push("The letter A");`	*clearly allowed*
`ts.push(new Integer(3));`	*clearly not allowed*
`to.push(new Integer(3));`	*oops, the compiler will incorrectly allow this*

- מסקנה: `FCStack<String>` הוא לא סוג של `FCStack<Object>`; זה לא אינטואיטיבי אבל נכון.

הכללה ויחס is-a (המשך)

- מסקנה: `FCStack<String>` הוא לא סוג של `FCStack<Object>`; זה לא אינטואיטיבי אבל נכון.
- ההשמה to = ts לא חוקית (שגיאת קומפילציה).
- לעומת זאת זה בסדר:

```
String [] as = new String[5];
Object [] ao = as;
```

- כלומר מערך של `String` הוא סוג של מערך של `Object`
- זאת הסיבה שלא יכולנו ליצור מערך גנרי:

content = new T[capacity] *// compile error*

טיפוסים נאים (raw types)

- מנגנון ההכללה נוסף לג'אווה מאוחר, ולכן היה צורך לאפשר שימוש במחלקות פרמטריות גם מקוד ישן שאין בו הכללות

```
class FCStack <T> implements Stack {...}
Stack <String> vs =
  new FCStack <String>();
Stack raw =                          same as Versioned<?>
  new FCStack ();                    same as Versioned<Object>
raw = vs;                            ok
vs = raw;                            "unckecked" compiler warning
```

- בשימוש בטיפוס נא, פרמטר הטיפוס מוחלף ב"גבול העליון" (בדרך כלל Object)

לדוגמא

- ראינו דוגמאות של המנשק Comparable בגירסא נאה raw
- אנחנו נעדיף את הגירסא הגנרית, שהשימוש בה הוא:

```
public class MyClass
      implements Comparable<MyClass> {
  public int compareTo(MyClass other) {
   }
}
```

- בצורה זאת מגדירים מחלקה שעצמיה ברי השוואה לעצמם, ומספקים שרות שמבצע את ההשוואה.
- אם רוצים אפשרות השוואה למחלקה כללית יותר, זה נעשה יותר מסובך (לא נעסוק בזה בינתיים).

מוזרויות

- בגלל שבג'אווה הכללה ממומשת באמצעות מנגנון המחיקה, בזמן ריצה אין זכר לפרמטר הטיפוס
- כלומר, בזמן ריצה אי אפשר להבחין בין עצם מטיפוס FCStack <String> ובין עצם מטיפוס FCStack <Integer>, ובפרט, בזמן ריצה נראה ששניהם מאותה מחלקה
- זה משפיע על בדיקת שייכות למחלקה (instanceof) (פרטים בהמשך), על יציקות של עצמים מוכללים, ועל שדות המסומנים static
- וזה מונע אפשרות לקרוא לבנאי על פי פרמטר טיפוס, כלומר

```
<T> void m(T x) { T y = new T(); ...}
```
illegal

סיכום generics

- מנגנון ההכללה מאפשר להימנע מיציקות בלי לשכפל קוד
- קוד שאין בו יציקות מפורשות ושאין בו טיפוסים נאים (ליתר דיוק, אם הקומפיילר לא הזהיר לגבי השימוש בטיפוסים נאים) הוא בטוח מבחינת טיפוסים (type safe)
- קוד כזה לא יכשל בביצוע יציקה בזמן ריצה: הבדיקות מועברות לזמן הקומפילציה
- השימוש בהכללה מסבך הצהרות על טיפוסים בגלל האינטראקציה הלא אינטואיטיבית בין טיפוסים מוכללים ובין יחס ה-is-a
- המימוש של הכללות בג׳אווה כולל מספר מוזרויות (ועוד לא דיברנו על כולן ...)

לנושא: 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 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

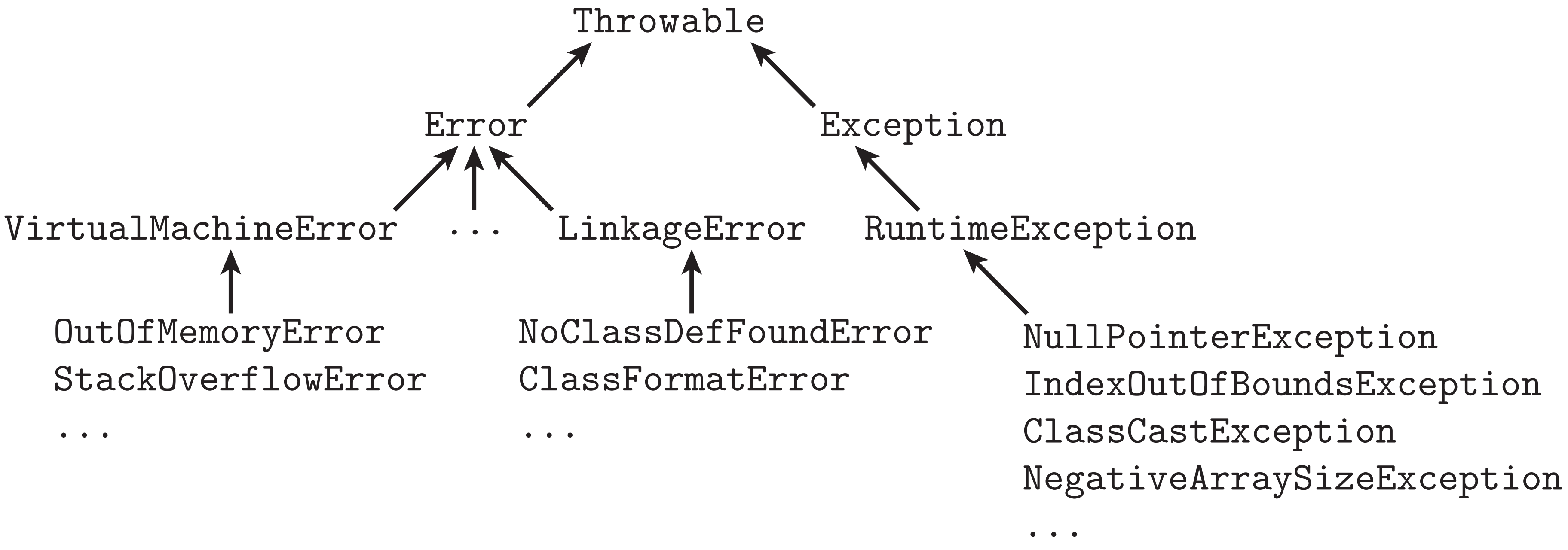
- בג'אווה, ההודעה על חריג מתבצעת באמצעות עצם רגיל שמייצג את החריג, את הכישלון של שירות כלשהו
- מכיוון שהחריג הוא עצם רגיל, בונים אותו בעזרת `new`
- הנוהג בג'אווה הוא לציין את הסיבה שגרמה לכישלון על ידי טיפוס חריג כמו `java.io.FileNotFoundException`
- ג'אווה מגדירה היררכיה של טיפוסים (מחלקות) עבור חריגים 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 בסביבת הריצה, שלא ניתן בדרך כלל להתאושש ממנה : מחסור בזיכרון, קבצי class חסרים או לא תקינים, וכדומה; התגובה הנכונה בדרך כלל היא להפסיק את ריצת התוכנית ולתקן את הסביבה
- Exception: חריגים פחות חמורים, שניתן במקרים רבים לטפל בהם כדי להמשיך בביצוע, והם נובעים מפגם בתכנית. מתחלקים לשתי קבוצות:
- RuntimeException הוא חריג שיכול לקרות כמעט בכל שירות: גישה למצביע null, כשלון ביציקה, חריגה מתחום מערך וכו'
- Exception שאינו RuntimeException מתרחש במצבים מוגדרים היטב, שניתן לתכנן מראש לקראתם.

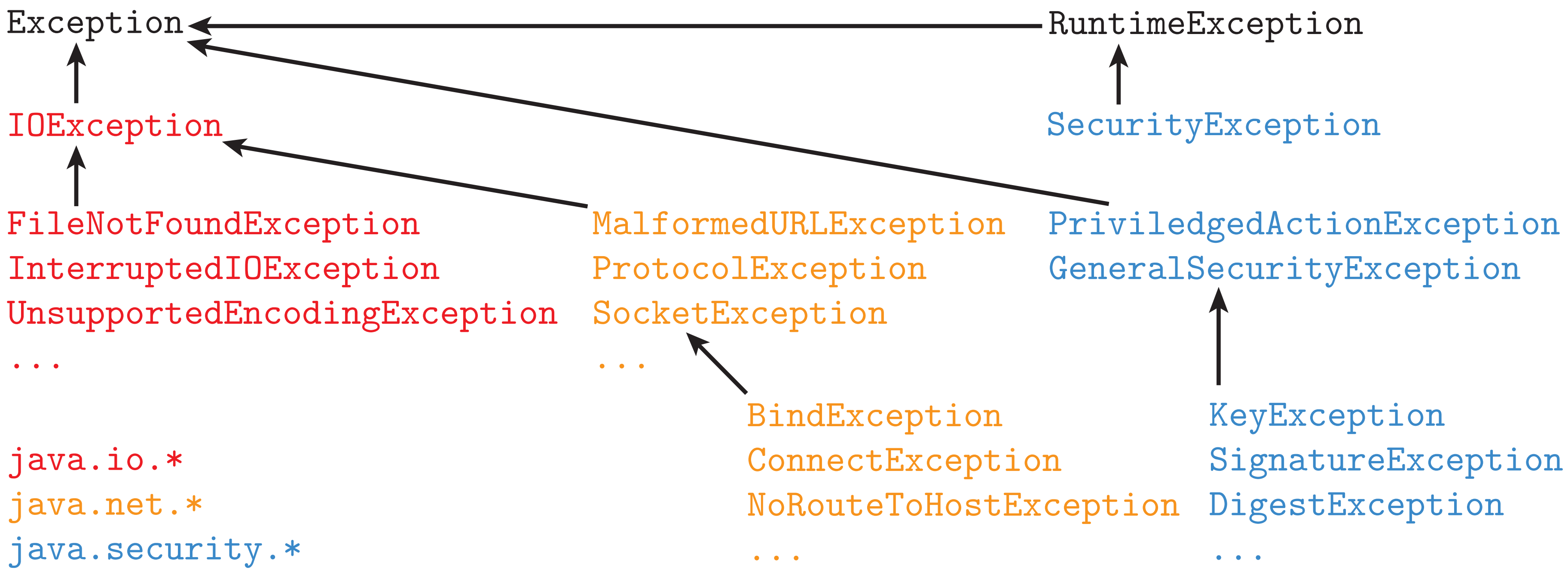
חריגים בחבילה java.lang

- חריגים מסוג Exception שאינם RuntimeException נקראים <u>חריגים נבדקים checked exceptions</u>
- חריגים משתי הקבוצות האחרות נקראים <u>חריגים בלתי נבדקים unchecked exceptions</u>



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 לזרוק חריג נבדק (checked exception) חייב להצהיר על האפשרות הזו בפסוק throws
- לקוח שקורא לשירות שהגדרתו כוללת הצהרה כזו חייב להתייחס לאפשרות שהשירות יכשל ויודיע על חריג
- טיפול אפשרי 1: כותרת השרות הלקוח (שבגוף שלו מופיעה הקריאה) מצהירה על אותו סוג חריג (פסוק throws)
- טיפול אפשרי 2: הקריאה לשירות היא מתוך 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 מועיל, למה יש חריגים שלא צריך 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 גבוה 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

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.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, 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

```
class BoundedVersionedString
  extends LinkedVersionedString {
  ...
  public void getVersion(i)
    throws DeletedVersionException {...}
```

- ניסינו לפתור את הבעיה על ידי השארת תנאי הקדם של getVersion על כנו, אבל הוספנו חריג כדי לטפל במקרה שלקוח מבקש גרסה שנמחקה כבר על ידי chop
- זה לא עובד! הוספת החריג החלישה בצורה אחרת את השירות, כי היא דורשת מהלקוח לטפל בחריג; את זה ג'אווה אוסרת תחבירית

ירושה וחריגים

- בג'אווה פסוק throws (ליתר דיוק להעדרו) הוא חלק מחוזה.
- שרות שמממש שרות מופשט (ממחלקה מופשטת שירש, או ממנשק שהוא מממש) 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 הקומפילר 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.

ירושה וחריגים - דוגמא

```
public class Base {
   public void doIt()
         throws Exception1, Exception2 { ..}
}
public class Sub extends Base {
   public void doIt()
         throws Exception1, Exception3 {..}
}
```

- הודעת שגיאה על Exception3

משפט assert

- משפט assert בג'אווה נועד לתעד ולוודא הנחות לגבי התכנית. (קיים החל מגירסא 1.4).
- ניתן להשתמש בו לצורך כתיבה ובדיקה של חוזים, אך לא באופן מלא.
- התחביר של משפט assert הוא משתי צורות:

```
assert <assertion>
assert <assertion> : <errorcode>
```

כאשר <assertion> הוא ביטוי בוליאני, ו <errorcode> הוא קוד שגיאה (מספר כלשהו) או מחרוזת.

- בביצוע רגיל של התכנית, משפטי assert לא מבצעים שום דבר.

משפט assert (המשך)

- ניתן לבקש שמשפטי assert יבוצעו (בכל התכנית, או במחלקות או חבילות נבחרות) ע"י פרמטר ea- ל JVM .
- מתוך eclipse בוחרים

Run -> Run... -> Arguments

ובתיבה "VM arguments" כותבים ea-

- כאשר משפטי assert מופעל, מחושב הביטוי הבוליאני. אם ערכו true לא נעשה שום דבר (התכנית ממשיכה להתבצע), אם ערכו false נוצר ונזרק החריג AssertionError . קוד השגיאה מועבר לבנאי של החריג.
- אם אין טיפול לחריג, התכנית נפסקת עם הודעת שגיאה שכוללת את קוד השגיאה שהופיע במשפט ה assert

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
- 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

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