תוכנה 1 בשפת Java

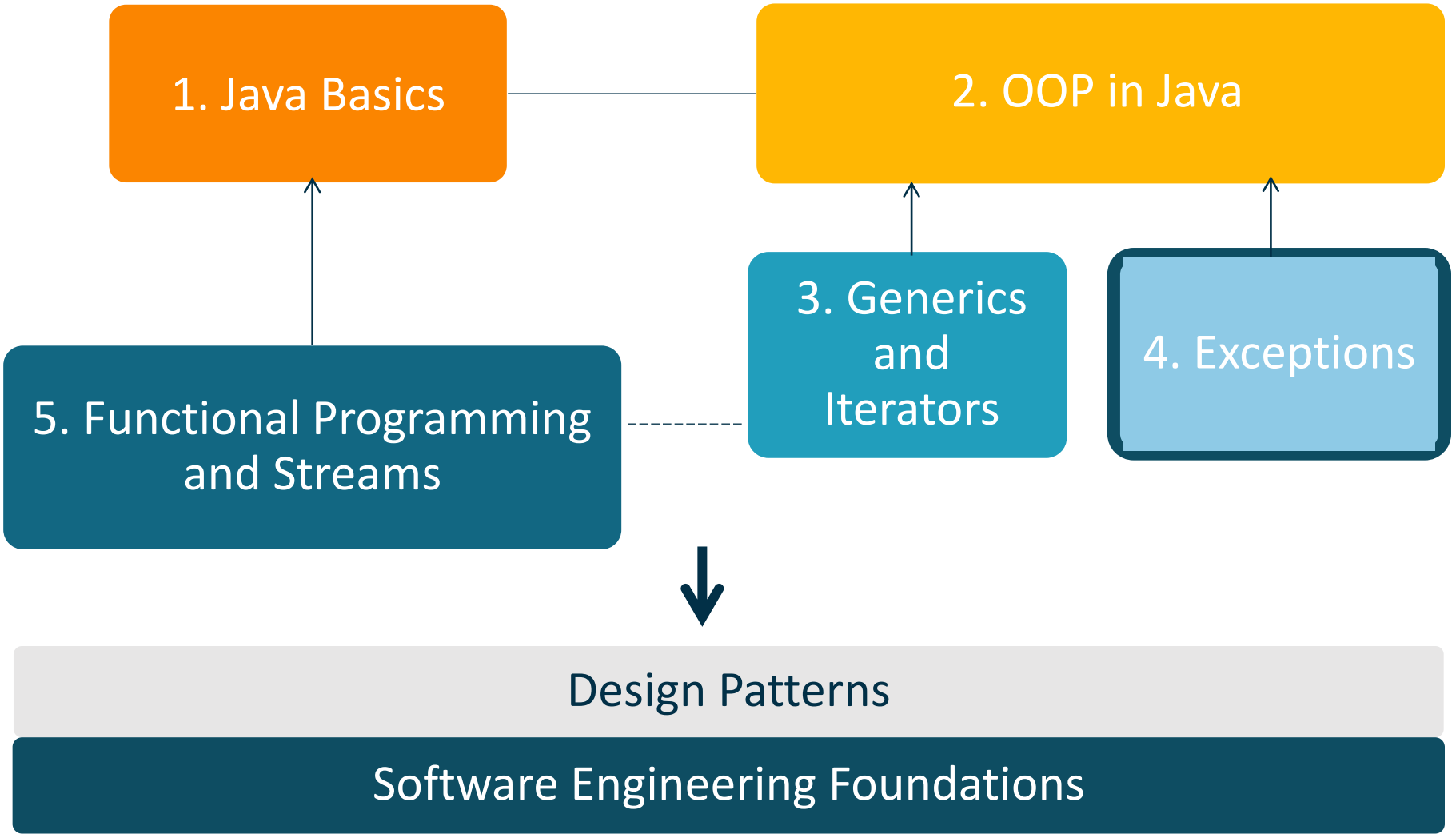
שיעור מספר 8: "יוצא מן הכלל" (הקדמה לחריגים)

מיכל קליינבורט

בית הספר למדעי המחשב

אוניברסיטת תל אביב

נושאי הקורס


1. Java Basics
2. OOP in Java
3. Generics and Iterators
4. Exceptions
5. Functional Programming and Streams
Design Patterns
Software Engineering Foundations

התוכנית להיום

- דוגמה
- מנגנון החריגים של Java

התכנית להיום

דוגמה

מנגנון החריגים של Java

מבוא לחריגים – hands on

- זריקת חריג:

```
public class Fib {
        public static long fib(int n) throws Exception  {
                if (n <= 0){
                        throw new Exception(
                    "Cannot calculate fib for a negative int");
                }
                long prev = 1, prevPrev = 0;
                if (n < 2){ return n; }
                while (n >= 2){
                        long tmp = prev+ prevPrev;
                        prevPrev = prev;
                        prev = tmp;
                        n--;
                }
                return prev;
        }
}
```

חריגים – hands on

שימוש בשירותים המצהירים על חריג

```
public static void main(String[] args){
        System.out.println(fib(5));
}
```

מכיוון ש fib מצהירה על חריג, הקוד לא יתקמפל אם לא נבחר באחד משני פתרונות:

- **הצהירי** – גם main תצהיר על חריג.
- **טפלי** – עלינו "לטפל" בחריג

שימו לב שבפועל, לא חייב להיזרק חריג (בדוגמת הקוד הזו הוא לא יזרק, אבל הקומפיילר מצריך או הצהרה או טיפול. כזכור, הוא עובד בצורה סטטית ולכן לא יודע איזה פרמטר נשלח בפועל.

"הצהירי"

- נצהיר על כך ש main עלולה לזרוק חריג

```
public static void main(String[] args) throws Exception {
        System.out.println(fib(5));
}
```

דוגמאות נוספות להצהרה על זריקה פוטנציאלית של חריג:

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

"טפלי"

- הפונקציה main תעטוף את הבלוק הבעייתי שעלול לזרוק חריג בבלוק try-catch

```
public static void main(String[] args){
        try{
                System.out.println(fib(5));
        }
        catch (Exception exp){
                System.out.println(”An exception has occured");
        }
        System.out.println("after try/catch block");
}
```

hands on – חריגים

- מה יקרה בשלא נזרק חריג?

```
public static void main(String[] args){
        try{
                System.out.println(fib(5));
        }
        catch (Exception exp){
                System.out.println(”An exception has occured");
        }
        System.out.println("after try/catch block");
}
```

```
5
after try/catch block
```

hands on – חריגים

- מה יקרה כשכן נזרק חריג?

```
public static void main(String[] args){
        try{
                System.out.println(fib(-5));
        }
        catch (Exception exp){
                System.out.println(”An exception has occured");
        }
        System.out.println("after try/catch block");
}
```

```
An exception has occured
after try/catch block
```

התוכנית להיום

דוגמה

מנגנון החריגים של Java

לא כל החריגים אותו דבר

- **חריגים (Exceptions)** מבטאים מצבים יוצאי דופן, מקרי קצה ומצבים בלתי צפויים בריצת התוכנית כגון: ארגומנטים שאינם חוקיים, בעיות ברשת התקשורת, קובץ שאינו קיים ועוד
- 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*)

הקשר לחוזים בין ספק-לקוח

- אם הלקוח קיים את תנאי הקדם (pre) לפונקציה **אבל הספק** אינו מצליח לקיים את תנאי האחר (post), אין לו אפשרות לבטל את הקריאה לשירות: היא כבר התבצעה
- הספק יכול לקיים את חלקו, או להשתמט, אבל **אינו יכול לבטל את העסקה**
- 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
- דוגמה לשירות כזה: Integer.parseInt

Integer.parseInt(String)

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 or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive 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 (בלומר, תנאי הקדם תמיד מתקיים), 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

טיפול בחריגים בג'אווה

- חריג יכול להיזרק ע"י פקודת **`throw`** (נראה בהמשך)
- קטע קוד אשר עלול לזרוק חריג יעטף ע"י הלקוח **`try` בבלוק**
- 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 בלוק זה
  - 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 התוכנית יסתיים

גרעיניות

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

> java AddArguments 1 two 3.0 4

parsing: 1

parsing: two

[two] is not an integer and will not be included in the sum.

parsing: 3.0

[3.0] is not an integer and will not be included in the sum.

parsing: 4

Sum = 5

ריבוי בלוקי catch

- לבלוק try אחד עשויים להיות כמה בלוקים של catch השייכים לו, עבור סוגים שונים של שגיאות שעשויות לקרות
- אם נזרקת שגיאה מתוך ה try, יבוצע לכל היותר בלוק catch אחד

```
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
}
```

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();
}
```

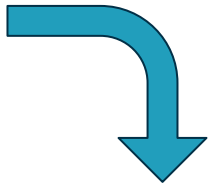
ברגע שנכנסנו לתוך בלוך ה try,
ה finaly יבוצע – לא משנה מה יקרה:

- לא קרתה שגיאה
- קרתה שגיאה וטופלה בבלוק catch
- קרתה שגיאה ולא טופלה בשום בלוק catch

try-with-resources

- טיפול במקרה פרטים של פתיחת משאבים המצריכים סגירה (מממשים את המנשק AutoCloseable)

```
FileReader fr = new FileReader(path);
BufferedReader br = new BufferedReader(fr);
try {
        return br.readLine();
}
finally {
        br.close();
        fr.close();
}
```



```
try (FileReader fr = new FileReader(path);
        BufferedReader br = new BufferedReader(fr)) {
        return          br.readLine();
}
```

מה עושה לקוח שמקבל חריג?

```
int compareTo(Comparable other) {
   IPoint other_point;
   other_point = (IPoint) other;
   if (this.x() > other_point.x())
         …
```

- המרת טיפוסים 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 בך שתהיה מובנת ללקוח שלו: מי שקרא ל-compareTo לא ביקש **להמיר טיפוסים** אלא **להשוות נקודות**

טיפוסים חריגים

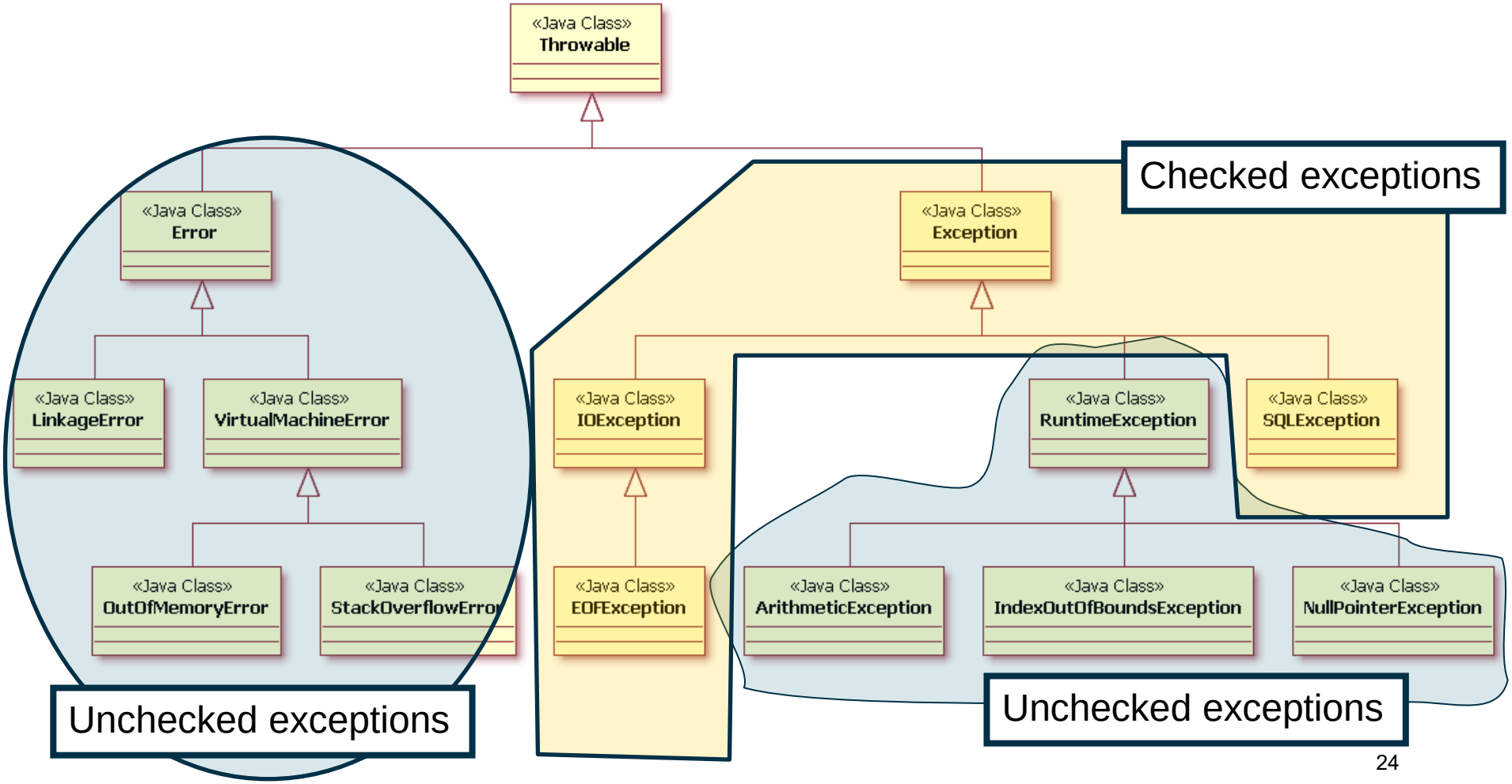
- בג'אווה, ההודעה על חריג מתבצעת באמצעות עצם רגיל שמייצג את החריג, את הבישלון של שירות כלשהו
- מכיוון שהחריג הוא עצם רגיל, בונים אותו בעזרת **`new`**
- הנוהג בג'אווה הוא לציין את הסיבה שגרמה לבישלון על ידי טיפוס חריג כמו **`java.io.FileNotFoundException`**
- ג'אווה מגדירה היררכיה של טיפוסים (מחלקות) 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

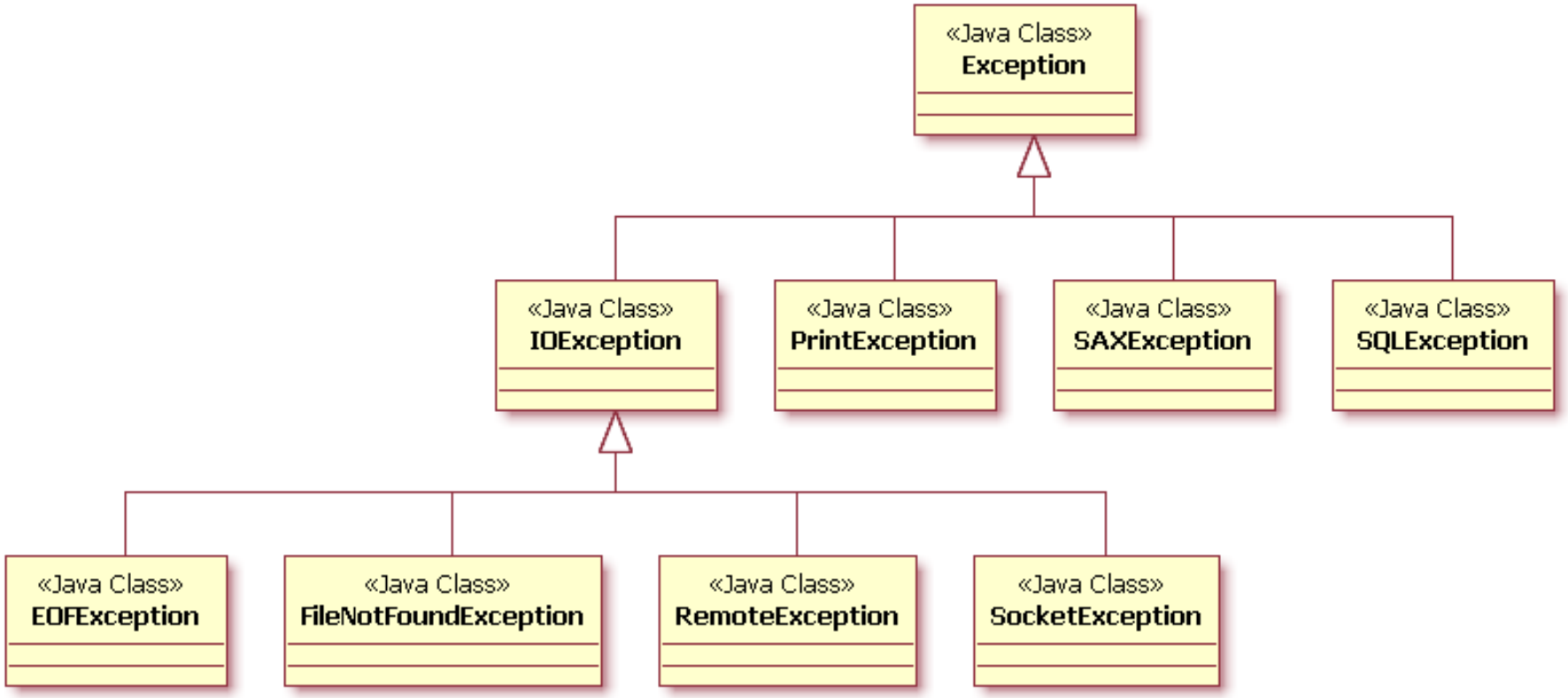
- **`Error`**: חריגים שמייצגים בעיה שלא ניתן בדרך כלל להתאושש ממנה:
  - בדרך כלל בעיה בסביבת הריצה: מחסור בזיכרון, קבצי class חסרים או לא תקינים, וכדומה; התגובה הנכונה בדרך כלל היא להפסיק את ריצת התוכנית ולתקן את הסביבה.
  - אולם, ניתן להגדיר חריגים מטיפוס Error כדי לבטא שבירה של הנחה לוגית (לדוגמא: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

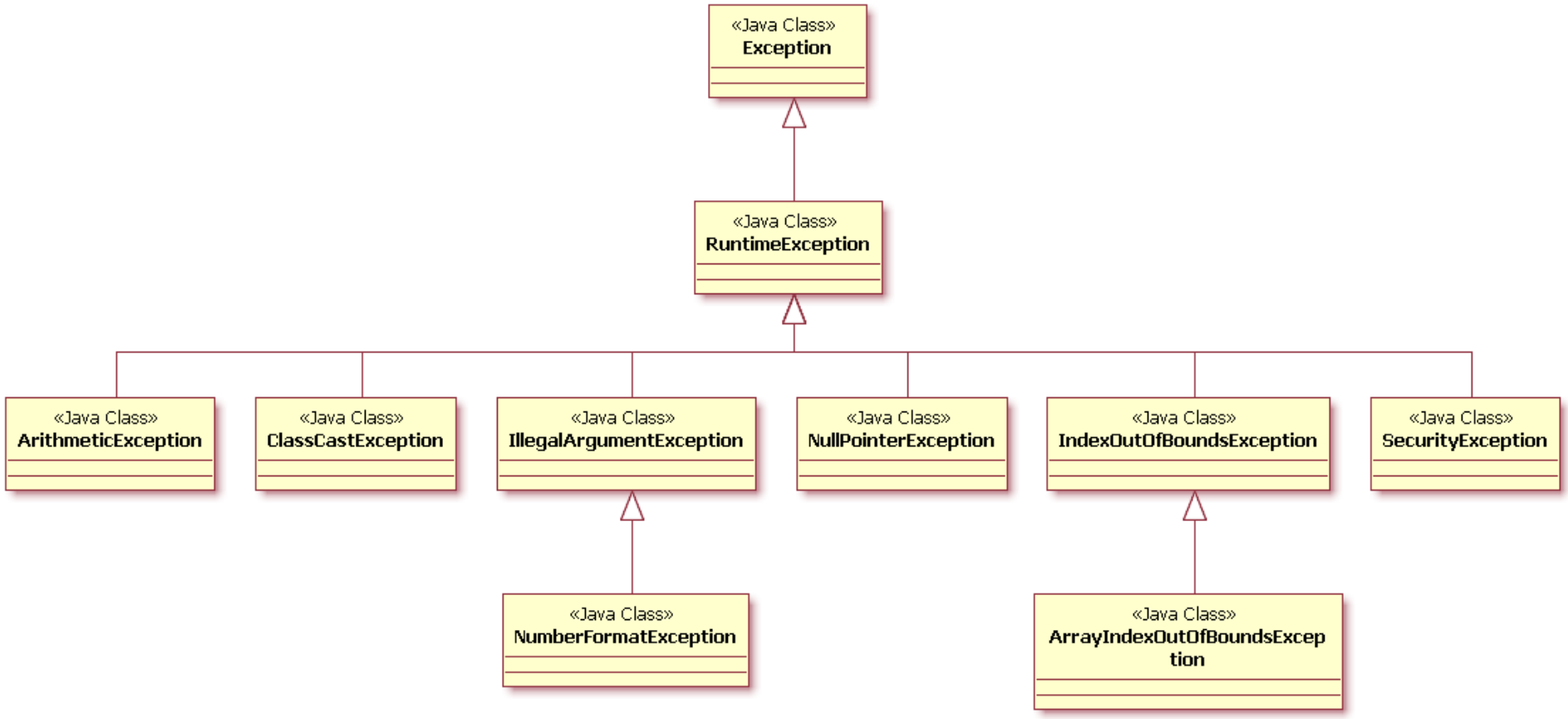
היררכיית שגיאות וחריגים (חלקית)



checked exceptions (רשימה חלקית)



unchecked exceptions (רשימה חלקית)



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 בישלון.

- אם זה מועיל, אז למה יש חריגים שלא **חייבים** להצהיר עליהם?

- מכיוון שחריגים מסוג **`RuntimeException`** או **`Error`** נזרקים 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

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 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 כלליות יותר (ואולי לא טובה): בג'אווה כל דבר הוא עצם

חריג בעצם

- 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 השפה של התוכניתן, ובושר 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 …");
}
```

שימוש לא מומלץ בחריגים

- 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

ירושה וחריגים: כללי 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