חריגות ובדיקות

כשאנחנו כותבים תוכנה, לא מספיק להתייחס למצבים הרגילים – צריך להתייחס גם למצבים ה**חריגים**, "לצפות את הלא-צפוי". בשפת ג'אבה ישנו מנגנון המאפשר למתודה להתריע על מצב חריג; בשפה המקצועית זה נקרא **לזרוק חריגה**. יש גם מנגנון תואם המאפשר למתכנת לזהות מצב חריג ולטפל בו; זה נקרא **לתפוס חריגה**.

איך זה עובד?

כשמתודה זורקת חריגה, המחשב מפסיק לבצע את המתודה ומעביר את החריגה למתודה שקראה לה; אם המתודה הזאת לא יודעת לטפל בחריגה, היא עוברת למתודה שקראה לה, וכן הלאה, עד שמתודה כלשהי **תופסת את החריגה** ומטפלת בה. אם אף מתודה לא תופסת את החריגה, התוכנית נעצרת ומדפיסה הודעת שגיאה. למשל, בתוכנית הבאה:

```
static void p(String s) {System.out.println(s);}
public static void main(String[] args) { p("main1"); a();
p("main2"); }
static void a() { p("a1"); b();    p("a2"); }
static void b() { p("b1"); c();    p("b2"); }
static void c() { Object x=null; x.toString(); }
```

המתודה c זורקת חריגה מסוג NullPointerException, שאף אחד לא תופס, ולכן התוכנית עוצרת ומדפיסה הודעת שגיאה. התוכנית לא תדפיס b2, a2, main2 כי לא לעולם לא תגיע לשם.

לעומת זאת, אם המתודה a מוחלפת ב:

```
p("a1");
try {
     b();
} catch (NullPointerException ex) {
     System.out.println("There was an exception: "+ex);
}
p("a2");
```

אז החריגה שנזרקה ב-c נתפסת ב-a, מודפסת הודעת שגיאה, והביצוע ממשיך אחרי סוף ה-catch.

זריקת חריגות

איך נוצרות חריגות? יש חריגות שנוצרות ע"י השפה: NullPointerException נוצרת כשמנסים לפנות למתודות של אובייקט שערכו null או שלא אותחל, ArithmeticException נוצרת למשל כשמנסים לחלק באפס, וכד'. אבל גם אתם המתכנתים יכולים ליצור חריגות – ואף כדאי מאד שתעשו זאת בכל פעם שהתוכנית שלכם נתקלת במצב חריג. למשל, אם מתודה שכתבתם מצפה לקבל ערכים חיוביים בלבד, עליכם לבדוק שכל הערכים אכן חיוביים, ואם אחד מהם שלילי – לזרוק חריגה מסוג IllegalArgumentException עם הודעה מתאימה. הנה דוגמה:

```
void setAge(int age) {
     if (age<0)
          throw new IllegalArgumentException("age must be
positive, but it is "+age);
     this.age = age;
}
```

כלל מקובל לטיפול בחריגות הוא "זרוק מוקדם, תפוס מאוחר", כלומר:

- המתודה הראשונה ששמה לב למצב חריג צריכה לזרוק חריגה - כדי שהמערכת לא תהיה במצב שגוי. לדוגמה, אם בנאי של מחלקה מסויימת מקבל פרמטר שלילי כשהמתודות של המחלקה יודעות להתמודד רק עם ערך חיובי, הכי טוב שהבנאי יזרוק חריגה – כך יהיה קל לזהות מי אשם בתקלה - מי שקרא לבנאי עם ערך שלילי. לעומת זאת, אם הבנאי ייצור עצם עם ערך שלילי והחריגה תיזרק מאוחר יותר על-ידי המתודות שמשתמשות בו, יהיה יותר קשה לדעת מי אשם.
- צריך לתפוס חריגה בנקודה שבה יודעים איך להתמודד עם המצב החריג. לדוגמה, אם המצב החריג הוא קובץ לא קיים, ישנן דרכים שונות להתמודד – אם הקובץ הוא קובץ-תצורה של התוכנית אז ההתמודדות הנכונה היא כנראה להשתמש בתצורת-ברירת-מחדל; אם הקובץ הוא קובץ חיוני יותר אז ההתמודדות הנכונה היא כנראה לסיים את התוכנית מייד. מי יודע מהי ההתמודדות הנכונה? בדרך-כלל מתודה כלשהי באיזור התוכנית הראשית. לכן עדיף שהחריגה תיתפס שם.

סוגי חריגות

שפת ג'אבה מאפשרת לזרוק ולתפוס סוגים שונים של חריגות. למה זה טוב? כי לפעמים מתודה יודעת להתמודד עם חריגה מסוג מסויים אבל לא יודעת להתמודד עם חריגה מסוג אחר; היא רוצה לתפוס רק את החריגות שהיא יודעת להתמודד איתן ולא חריגות אחרות. החריגות בג'אבה מסודרות במבנה מדרגי של ירושות:

- `Throwable` – מחלקת הבסיס של כל המדרג.
  - `Error` – בעיה חמורה שמתכנת סביר לא אמור לדעת להתמודד איתה, למשל:
    - `OutOfMemoryError` – נגמר הזיכרון;
    - `NoClassDefFoundError` – המחשב לא מצא הגדרה של מחלקה.
  - `Exception` – חריגה שמתכנת סביר אמור לדעת להתמודד איתה. יש שני סוגים עיקריים:
    - `RuntimeException` – חריגה **שאינה נבדקת** ע"י הקומפיילר – המתכנת צריך לוודא שהיא לא תקרה מלכתחילה. למשל:
      - `NullPointerException`
      - `ArithmeticException`
      - `IllegalArgumentException`
    - כל שאר החריגות **נבדקות** ע"י הקומפיילר. הכי שימושיות בקבוצה זו הן:
      - `IOException` – חריגה כלשהי הקשורה לקלט או פלט.
        - `FileNotFoundException` – הקובץ לא נמצא.
        - `UnsupportedEncodingException` – קידוד-התוים בקובץ לא מוכר.

בהמשך נסביר מה המשמעות של "בדיקה ע"י הקומפיילר". אבל קודם בואו נראה איך עובד מנגנון התפיסות לפי סוגים.

נניח שבמתודה `a` למעלה, במקום לתפוס `NullPointerException`, אנחנו תופסים `ArithmeticException`. במקרה זה, החריגה שנזרקת מהמתודה `c` אינה נתפסת, כי היא מסוג אחר. התוכנית עוצרת ומדפיסה הודעת שגיאה. מה קורה אם במתודה `a` אנחנו תופסים `RuntimeException`? במקרה זה החריגה שנזרקת מהמתודה `c` כן נתפסת, כי היא מקרה פרטי של `RuntimeException`.

ניתן לכתוב כמה פסקאות-תפיסה בזו אחר זו: המחשב יבדוק אותן לפי הסדר ויעצור ברגע שימצא פסקה מתאימה. למשל, אם נכתוב:

```
try {
```

```
    b();
} catch (NullPointerException ex) {
    System.out.println("Did you use a null object?"+ex);
} catch (ArithmeticException ex) {
    System.out.println("Did you divide by zero?"+ex);
} catch (RuntimeException ex) {
    System.out.println("What did you do? "+ex);
}
```

אז פניות לאובייקטי-נול יגיעו לפיסקה הראשונה, חלוקה באפס תגיע לפסקה השניה, כל חריגה אחרת מסוג RuntimeException או אחד הצאצאים שלה (כגון IllegalArgumentException) תגיע לפיסקה השלישית, וכל חריגה אחרת לא תיתפס כלל.

חידה: מה צריך להוסיף למתודה a כך שהיא תתפוס את **כל** סוגי החריגות והשגיאות?

אתם יכולים ליצור סוגי-חריגות חדשים לפי הצורך. למשל, אם יש לכם מתודה שמקבלת פרמטר "צבע" והיא קיבלה צבע לא חוקי, אולי תרצו לזרוק חריגה מסוג "צבע לא חוקי". אפשר להגדיר אותה למשל כך:

```
public class IllegalColorException extends RuntimeException {
    public IllegalColorException() {super();}
    public IllegalColorException(String message)
{super(message);}
}
```

חריגות נבדקות ולא-נבדקות

מה המשמעות של "בדיקה ע"י הקומפיילר"? - אם מתודה עלולה לזרוק חריגה שנבדקת (checked exception), אז היא צריכה **להצהיר** עליה בכותרת שלה. המתודה הקוראת צריכה או לתפוס את החריגה או להצהיר עליה. אם היא החליטה להצהיר עליה, אז המתודה הקוראת לה צריכה לתפוס את החריגה או להצהיר עליה, וכן הלאה. לעומת זאת, אם מתודה עלולה לזרוק חריגה לא-נבדקת (unchecked exception), אז היא לא צריכה להצהיר עליה (אבל היא רשאית לתפוס אותה).

לדוגמה, נניח שהמתודה c מהדוגמה למעלה זורקת, במקום NullPointerException, חריגה מסוג FileNotFoundException. במקרה זה הקוד לא יתקמפל – הקומפיילר ידרוש שנצהיר על כך בראש המתודה c:

```
static void c() throws FileNotFoundException { … }
```

עכשיו המתודה b לא מתקמפלת – הקומפיילר מזהה שהיא עלולה לזרוק חריגה נבדקת, ודורש שנתפוס אותה או נצהיר עליה. אם נבחר להצהיר עליה, זה ייראה כך:

```
static void b() throws FileNotFoundException { … }
```

מה דעתכם, האם עכשיו הקוד יתקמפל? --לא! עכשיו המתודה a לא מתקמפלת, הקומפיילר מזהה שהיא עלולה לזרוק חריגה נבדקת, ושוב אנחנו צריכים להחליט עם לתפוס אותה או להצהיר עליה. אם נתפוס אותה (כמו שעשינו למעלה) הקוד יתקמפל, ואם נצהיר עליה – המתודה main לא תתקמפל. הפעם, בין אם נבחר להצהיר על המתודה בכותרת של main או לתפוס אותה, הקוד סוף-סוף יתקמפל.

חידה: מה יקרה אם המתודה b תצהיר, לא על FileNotFoundException אלא על חריגה מסוג כללי יותר -- IOException? (בידקו ותראו).

כפי ששמתם לב, התהליך של "פעפוע" חריגות נבדקות הוא ארוך ומעייף. בקהילת מתכנתי ג'אבה, הדעות חלוקות אם המנגנון הזה של חריגות נבדקות הוא טוב או רע. יש מתכנתים רבים שחושבים שהמנגנון הזה מיותר, ועדיף לזרוק רק חריגות לא נבדקות. למרות זאת צריך להכיר את המנגנון של חריגות נבדקות, בעיקר כשעובדים עם קבצים, שכן המתודות הסטנדרטיות של ג'אבה לעבודה עם קבצים זורקות חריגות נבדקות.

מה עושים עם חריגה אחרי שתופסים אותה?

יש כמה דברים שמקובל לעשות בפיסקת catch כשתופסים חריגה:

(א) אם אנחנו בתוכנית שיכולה לכתוב למסך, אפשר פשוט להדפיס את החריגה (ex), או רק את הודעת השגיאה שלה (ex.getMessage).

(ב) המתודה ex.printStackTrace מדפיסה למסך את כל שרשרת הקריאות, החל מהמתודה הראשית ועד לנקודה שבה נזרקה החריגה. זה מאד שימושי כדי לזהות מי בדיוק אשם בשגיאה.

(ג) בתוכניות גדולות יותר, מקובל שלא לכתוב למסך כי לא בטוח שיהיה מישהו שיסתכל על המסך כל הזמן. במקום זה, מקובל לכתוב את הודעת השגיאה לקובץ יומן (log). בהמשך נלמד (אולי) איך לנהל קבצי יומן. דרך אפשרית לרישום שגיאה ביומן היא:

```
Logger.getGlobal().severe(ex.toString());
```

(ד) מה עושים אם רוצים לכתוב ביומן את כל שרשרת הקריאות? לשם כך צריך להפוך את הפלט של המתודה printStackTrace למחרוזת. איך עושים את זה? משתמשים בגירסה של printStackTrace המקבלת ארגומנט מסוג PrintWriter, ומעבירים לה PrintWriter שעוטף StringWriter, למשל:

```
StringWriter w = new StringWriter();
ex.printStackTrace(new PrintWriter(w));
Logger.getGlobal().severe(w.toString());
```

(ה) לפעמים, מתודה תופסת חריגה ולא יודעת מה לעשות איתה; במקרה כזה, היא צריכה לכתוב הערה ביומן ולזרוק את אותה חריגה הלאה, למתודה הקוראת. לשם כך כותבים, בסוף פיסקת ה-catch, את המשפט throw ex. אפשרות נוספת היא לעטוף את החריגה ex בחריגה מסוג אחר. למשל, נניח שיש לנו מתודה שמנסה לקרוא קובץ הגדרות. המתודה יכולה להיתקל בהרבה תקלות שונות, כגון: קובץ לא קיים, קובץ לא תקין, לאחד הפרמטרים בקובץ יש ערך לא תקין, וכו'. המתודה יכולה לתפוס את כל התקלות הללו, לכתוב הערה מתאימה ביומן, ואחר-כך לזרוק חריגה אחת כללית יותר, למשל:

```
catch (FileNotFoundException ex) {
     Logger.getGlobal().warning(ex.toString());
     throw new ConfigurationFileException
          ("Cannot read configuration file", ex);
}
```

הפרמטר השני מאפשר לדווח על הסיבה vשגרמה לחריגה. כדי לקרוא אותו משתמשים במתודה getCause.

סגירת משאבים

כשפותחים קובץ, לקריאה או לכתיבה, צריך לסגור אותו אחרי השימוש. אם לא סוגרים קובץ שנפתח לקריאה – הוא עלול להישאר נעול ותוכניות אחרות לא יוכלו לערוך אותו. אם לא סוגרים קובץ שנפתח לכתיבה – הוא גם עלול להישאר נעול, ובנוסף לכך ייתכן שהטקסט שכתבנו לתוכו לא יגיע אליו.

אבל מה קורה אם, לפני שהגענו לפעולת הסגירה, נזרקה חריגה? איך נוכל לוודא שהקובץ ייסגר כראוי? הפתרון בשפת ג'אבה 8 הוא לאתחל את הקובץ בתוך פקודת ה-try. למשל, לקריאת קובץ:

```
try (Stream<String> lines = Files.lines(theFile)) {
     lines.forEach(line -> System.out.println(line));
} catch (IOException e) {
     e.printStackTrace();
}
```

ולכתיבת קובץ:

```
try (PrintWriter writer = new
PrintWriter(Files.newBufferedWriter(theFile))) {
     writer.println("Hello world");
} catch (IOException e) {
     e.printStackTrace();
}
```

(כאשר theFile הוא משתנה מסוג Path).
כאשר הקובץ נפתח בתוך ה-try, המחשב יודע לסגור אותו בכל מצב – בין אם ה-try הסתיים בלי חריגה, או עם חריגה שנתפסה ב-catch, או עם חריגה שלא נתפסה כלל.

יש שיטה ישנה יותר המאפשרת להבטיח שפעולה מסויימת תתבצע לבסוף, בין אם ה-try הסתיים בלי חריגה או עם חריגה. השיטה היא להוסיף בלוק finally, למשל בקריאת קובץ:

```
Stream<String> lines = null;
try {
     lines = Files.lines(theFile);
     lines.forEach(line -> System.out.println(line));
     lines.close();
} catch (IOException e) {
     e.printStackTrace();
} finally {
     lines.close();
}
```

הבעיה בשיטה זו היא, שהיא לא מגנה מפני חריגות בפעולת ה-close עצמה. השיטה של איתחול המשאבים בתוך ה-try היא גם קצרה יותר וגם בטוחה יותר. כדאי להשתמש ב-finally רק כשצריך לבצע פעולות ניקיון נוספות, שאינן קשורות לסגירת משאבים.

הצהרות פנימיות - assert

המנגנון של זריקת חריגות נועד להתריע של שגיאה של מישהו אחר, שמשתמש במתודה שלכם. אבל לפעמים אתם רוצים לבדוק, שאתם עצמכם לא עשיתם שגיאה. לשם כך, אתם רוצים להיות בטוחים שתנאי מסויים מתקיים במקום מסויים בקוד. לשם כך אפשר להשתמש בפקודת assert.

למשל, נניח שאתם עושים חישוב מסובך והוא נכשל כי מספר מסויים x הוא שלילי כשלמעשה הוא צריך להיות חיובי. אתם רוצים לברר איך הגיע ערך שלילי למשתנה שלכם. אז אתם מוצאים נקודה בקוד שבה, לדעתכם, המשתנה חייב להיות חיובי, וכותבים שם:

```
assert x>=0;
```

מה יקרה אם המחשב בכל-זאת יגלה שבנקודה הזאת קטן מאפס? במצב רגיל, לא יקרה כלום! במצב רגיל המכונה הוירטואלי של ג'אבה מסירה אוטומטית את כל שורות ה-assert, כדי לחסוך בזמן-ריצה. כדי להשאיר את השורות האלו בקוד, צריך להעביר ל-java את אחד משני הפרמטרים הבאים:

```
-ea      /      -enableassertions
```

[כדי לעשות זאת באקליפס, לחצו עם הכפתור הימני על שם המחלקה, בחרו Run As, בחרו Run Configurations, בחרו Arguments, לכו ל VM Arguments, וכיתבו שם מינוס (-) ואז ea].

ואז, אם הערך של x יהיה שלילי, תיזרק חריגה מסוג AssertionError.

אפשר להוסיף פרמטר לפעולת assert כך שהחריגה תכיל מידע נוסף, למשל אם תכתבו:

```
assert x>=0 : x;
```

החריגה תכיל את הערך של x בנקודה זו – זה יקל עליכם לדעת מאיפה הגיעה השגיאה.

אתם יכולים להוסיף עוד ועוד שורות assert לקוד שלכם, עד שתגלו את השגיאה. ברגע שסיימתם לדבג את התוכנית שלכם, פשוט הסירו את הפרמטר ea והקוד ירוץ מהר, כאילו שהבדיקות לא קיימות.

בדיקות אוטומטיות – Junit

קיימים כלים המאפשרים לכם לבדוק את הקוד שלכם באופן אוטומטי. במקום להריץ מתודות, להדפיס את התוצאה שלהן ולבדוק בעין שהתוצאה נכונה, אתם מגדירים סדרה של בדיקות, ולכל בדיקה מגדירים את התוצאות הצפויות, ונותנים למחשב לעשות את העבודה. בסוף הבדיקה תקבלו דיווח כמה בדיקות עברו, כמה בדיקות נכשלו ואיזה בדיקות נכשלו.

הכלי הנפוץ ביותר לביצוע בדיקות מסוג זה בג'אבה הוא Junit. זה כלי חיצוני שאינו נחשב חלק רשמי משפת Java, אבל הוא כל-כך שימושי שבסביבות עבודה כמו אקליפס מאפשרים לכם ליצור מחלקות של Junit באופן אוטומטי (File – New – Junit test case). בכל מחלקה של Junit צריכה להיות מתודה אחת או יותר המתוייגת בתג @Test. כל מתודה כזאת מייצגת בדיקה אחת. לדוגמה, הנה התחלה של מחלקת-בדיקות עבור המחלקה "מונום" שראינו באחד השיעורים הקודמים:

```
class UnitestMonom {
     @Test void testToString() {
          Monom m = new Monom(5,2);
          assertThat(m.toString(), is("5.0x^2"));
     }
     @Test void testDegree() {
          Monom m = new Monom(5,2);
          assertThat(m.getDegree(), is(2));
     }
}
```

כשמריצים מחלקה ב JUnit, המערכת מריצה את כל המתודות המסומנות ב@Test, ומדווחת כמה מהן הסתיימו בהצלחה וכמה הסתיימו בשגיאה (באקליפס, אם כל הבדיקות הסתיימו בהצלחה הדו"ח יהיה בצבע ירוק, ואם אחת או יותר נכשלה הדו"ח יהיה בצבע אדום).

בנוסף לבדיקה is יש בדיקות רבות נוספות שאפשר לבצע - לפרטים ראו בסרטון המצורף.

איך בודקים שמתודה זורקת חריגה? – משתמשים ב-assertThrows באופן הבא:

```
void testSetPowerNegative() {
     assertThrows(IllegalArgumentException.class,
          () -> m.setPower(-2));
```

}

ב Junit יש עוד הרבה אפשרויות שמקלות על ביצוע בדיקות. למשל, אם רוצים להתעלם מבדיקה מסויימת באופן זמני כדי להתמקד בבדיקות האחרות, אפשר לשים לפני התיוג @Test את התיוג @Disabled. אפשר לקבוע timeout לבדיקה כדי לוודא שהיא לא נמשכת יותר מדי זמן. אפשר להגדיר חבילת בדיקות – TestSuite – כך שנוכל להריץ הרבה בדיקות בלחיצת-כפתור אחת.
לפרטים ראו בסרטון המצורף

מקורות

- Cay Horstmann, “Core Java for the Impatient”, chapter 5.
- Why “throw early, catch late”?
  https://softwareengineering.stackexchange.com/a/231064/50606
- Are checked exceptions good or bad?
  https://www.javaworld.com/article/3142626/core-java/are-checked-exceptions-good-or-bad.html
- Unit testing with Junit
  https://www.youtube.com/watch?v=F61aUno8qHk
- Junit 5 User Guide
  http://junit.org/junit5/docs/current/user-guide/

סיכם: אראל סגל-הלוי.