Listeners & Extensions Part 2

הפוסט הזה יהיה תמציתי, יחסית. יחד עם זאת חסר בהיקפו.

הסיבה שהפוסט יהיה תמציתי היא שאחרי שדנו לעומק ברעיון ההרחבות וה- listeners בחלק הראשון של הפוסט, והבנו שבכל Framework שבנוי בצורה טובה יש רכיב כזה, כל שנותר לנו הוא לראות מס' דוגמאות מימוש של listener ב- Junit. אנחנו נשתמש ב- Junit 5 לצורך הדוגמא, אך דוגמאות נוספות קיימות גם ב- Junit 4, TestNG, וכמובן גם בשפות פיתוח נוספות וב- frameworks נוספים.

מאידך, הסיבה שהפוסט לא יהיה חסר בהיקפו (לפחות בעיני) היא שמבנה ההרחבות ב- Junit 5 הוא נושא כל כך רחב, ומכיל כל כך הרבה תתי נושאים, שיהיה מאוד קשה להקיף אותו בצורה ממצה בפוסט אחד. כותבי Junit 5 עשו עבודה טובה בלאפשר גמישות למשתמשים שלהם.

החדשות הטובות הן, שמספיק לנו ידע בסיסי בהרחבות של Junit 5 על מנת לשפר את תשתיות האוטומציה שלנו בצורה משמעותית.

שנתחיל?

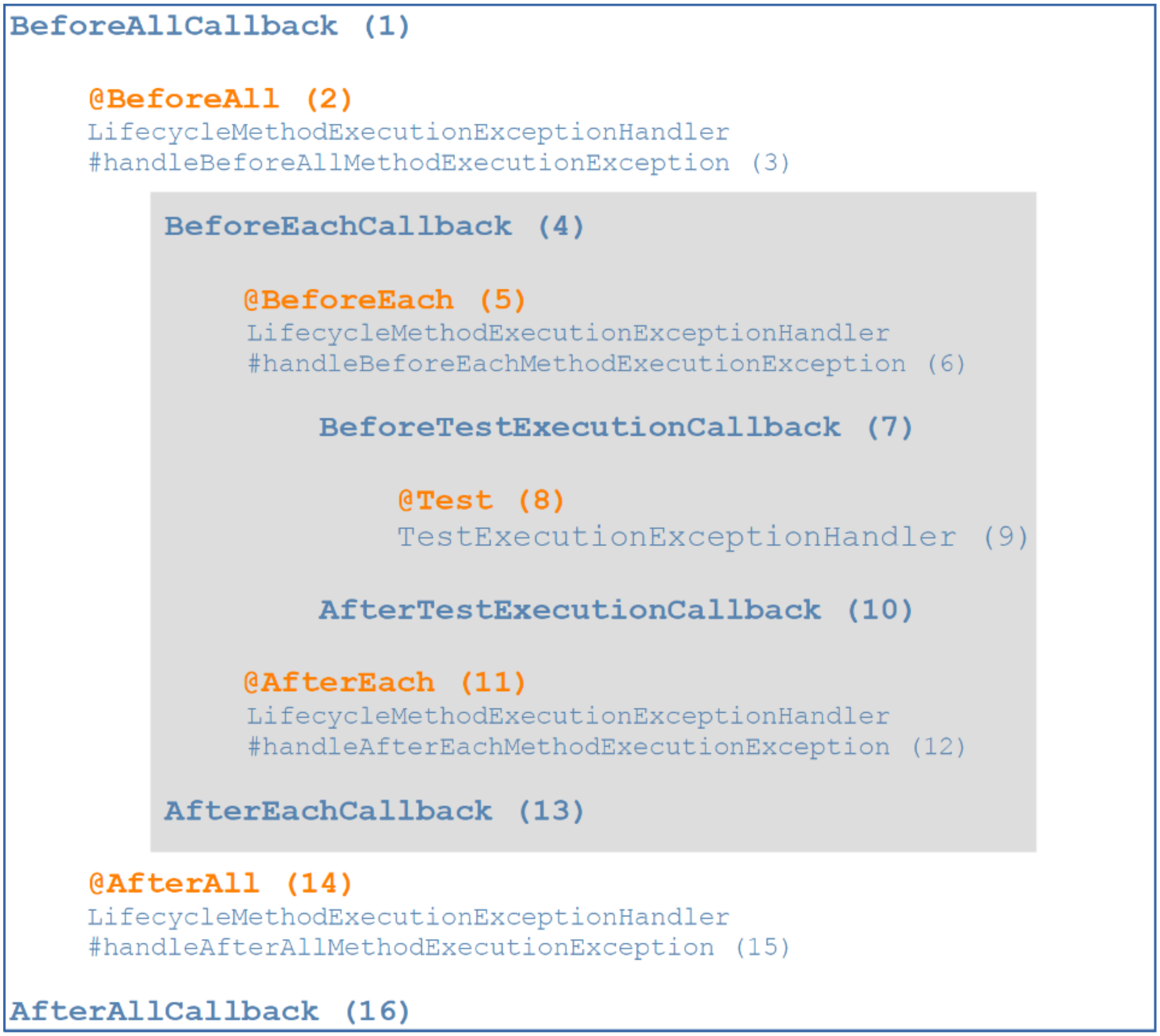
אם לא קראתן את החלק הראשון, אתן מוזמנות לחזור ולעיין בו, הוא חשוב להבנת הרעיון המארגן של הנושא.

בקצרה נאמר שעל ידי Extensions & Listeners אנחנו מסוגלים לעמוד ב- 2 עקרונות SOLID חשובים (עקרונות חשובים כאשר כותבים Object Oriented Programming):

א. לכל יחידת קוד (class, method) צריך להיות **רק תפקיד אחד**.

ב. יחידות קוד צריכות להיות סגורות לשינויים מצד אחד (כדי להגביר יציבות ולמנוע תקלות), ולאפשר הרחבות, מצד שני (כדי לאפשר גמישות לשינויים והתפתחויות במהלך הקידוד).

ל- Junit יש רצף הפעלה. רצף ההפעלה של Junit מתואר בסכימה הבאה. היא לקוחה מה- user guide של Junit וניתן לראות אותה כאן:



BeforeAllCallback (1)

@BeforeAll (2)
LifecycleMethodExecutionExceptionHandler
#handleBeforeAllMethodExecutionException (3)

BeforeEachCallback (4)

@BeforeEach (5)
LifecycleMethodExecutionExceptionHandler
#handleBeforeEachMethodExecutionException (6)

BeforeTestExecutionCallback (7)

@Test (8)
TestExecutionExceptionHandler (9)

AfterTestExecutionCallback (10)

@AfterEach (11)
LifecycleMethodExecutionExceptionHandler
#handleAfterEachMethodExecutionException (12)

AfterEachCallback (13)

@AfterAll (14)
LifecycleMethodExecutionExceptionHandler
#handleAfterAllMethodExecutionException (15)

AfterAllCallback (16)

מניתוח זריז של רצף ההפעלה רואים מתודות בצבע כתום:

```
@BeforeAll, @BeforeEach, @Test, @AfterEach, @AfterAll
```

ומתודות בצבע כחול:

```
BeforeAllCallback, BeforeEachCallBack, BeforeTestExecutionCallback,
AfterTestExecutionCallback, AfterEachCallback, AfterAllCallback
```

המתודות בצבע **כתום** אלו הן מתודות שמופיעות כחלק ממחלקת הבדיקות עצמה, לדוגמא, המחלקה הבאה:

```
public class TestClass {

    @BeforeAll
    public static void generalSetup(){
        System.out.println("in Before All");
    }

    @BeforeEach
    public void setup(){
        System.out.println("in Before each");
    }

    @Test
    public void test1(){
        System.out.println("in test1");
    }

    @Test
    public void test2(){
        System.out.println("in test2");
    }

    @AfterEach
    public void tear(){
        System.out.println("in After each");
    }

    @AfterAll
    public static void tearAll(){
        System.out.println("in After all");
    }

}
```

הרצה של המחלקה הזו תייצר את התוצאה הבאה:

```
In Before all
In Before each
In test 1
```

```
In After Each

In Before each

In test 2

In After Each

In After All
```

אין חובה לכתוב את שמות המתודות כפי שכתבנו אנחנו. זה שם לצורך הנוחות בלבד. האנוטציה שמעליה (@) מכתיבה את ההתנהגות שלה.

שימו לב ש- `BeforeAll` ו- `AfterAll` הן מתודות סטטיות. `BeforeAll` תרוץ לפני שהמחלקה נבנתה (כלומר, לפני ה- constructor). בשלב זה המחלקה איננה אובייקט ולכן מתודה חייבת להיות סטטית.

`AfterAll` תרוץ לאחר שכלל הטסטים יסיימו ריצה, ולא יהיה לנו עוד צורך במחלקה. היא גם סטטית.

ועכשיו מתחיל הכיף...

רצף המתודות בצבע הכחול הן הרחבות, ולא חלק ממחלקת הטסט עצמה. כלומר, נוכל לכתוב מחלקת הרחבה שתממש כל אחד מהמתודות ברצף הכחול.

כמו בחלק א' של הפוסט, גם כאן, נרצה לשמור על עקרונות SOLID, ולכן נרצה לבנות הרחבה לנושא אחד בלבד.

נניח נרצה לבנות הרחבה שתטפל בנושא הדו"חות. לדו"חות יש כמה שלבים במהלך הטסטים:

א. יצירה של אובייקט הדו"ח עצמו - פעולה המתרחשת **פעם אחת** בכל הריצה האוטומטית
ב. יצירה של אובייקט מסוג טסט **בכל פעם** שטסט מתחיל
ג. כתיבה לדו"ח במהלך הטסט
ד. החלטה האם הטסט עבר או נכשל **בכל סיום** של טסט.
ה. סגירה של אובייקט מסוג טסט **בכל פעם** שטסט מסתיים
ו. סגירה של אובייקט מסוג דו"ח **בסיום הבדיקות**.

אפשר לראות שלמעט שלב ג', כל שאר השלבים הם אוטומטיים ומתאימים להרחבה מהסוג שעליו אנחנו דנים.

בפוסט כמובן לא נשלב כתיבה אמיתית לדו"ח, נשאיר את נושא הדו"חות לפוסט אחר, אבל נוכל להראות את המבנה.

אז, כמו בהרחבות אחרות אנו עומדים לבצע שני שלבים:

א. כתיבת מחלקת ההרחבה

ב. רישום מחלקת הרחבה על מנת שתוכל לפעול.

נכתוב את המחלקה:

```
public class ReportExtension implements
       AfterAllCallback,
       AfterEachCallback,
       BeforeAllCallback,
       BeforeEachCallback {

   @Override
   public void beforeAll(ExtensionContext context) throws Exception {
       Reporter.createNewReportObject();
   }
   @Override
   public void beforeEach(ExtensionContext context) throws Exception {
       Reporter.createNewTest(context.getTestMethod().get().getName());
   }

   @Override
   public void afterEach(ExtensionContext context) throws Exception {
       if (context.getExecutionException().isPresent()) {
           Reporter.setResult("Fail");
       }
       else{
           Reporter.setResult("Pass");
       }
   }

   @Override
   public void afterAll(ExtensionContext context) throws Exception {
       Reporter.finishReportAndCreateHTML();
   }

}
```

והרישום של המחלקה במחלקת הבדיקות שכתבנו מקודם תהיה:

```
@ExtendWith({ReportExtension.class})
Public class TestClass {

    @BeforeAll
    public static void generalSetup(){
        System.out.println("in Before All");
    }


...

}
```

אנחנו יכולים לכתוב מס' הרחבות ולשלבן בתוך המחלקה. נניח שכתבנו 3 הרחבות:

- הרחבה שכאשר הטסט נכשל תקח ותשמור צילום מסך `ScreenShotExtension`
- הרחבה שתבצע הקלטת וידאו של כל טסט וטסט `ScreenRecorderExtension`
- הרחבה שמנהלת את תהליך הכתיבה לדו"ח (ההרחבה שכתבנו לפני רגע) `ReportExtension`

אז נוכל לרשום את כל ההרחבות באופן הבא, וכולן יפעלו, על פי התזמון שהוגדר להן במהלך הבדיקות:

```
@ExtendWith({ReportExtension.class, ScreenShotExtension.class,
ScreenRecorderExtension.class})
Public class TestClass {
     ...
}
```

על אובייקט context, בקצרה

הפירוש המילולי של context הוא הקשר, או מודעות. מכיוון שאנחנו מפעילים הרחבות, והרחבה היא מחלקה נפרדת מהטסט, לפעמים נרצה לקבל ולדעת נתונים על הטסט. לדוגמא:

- מה שם מחלקת הטסטים שרצה?
- מה שם הטסט שרץ?
- האם התרחש exception במהלך הטסט (וזה אומר שהטסט נכשל)
- אילו פרמטרים השתמשנו בטסט?

כל המידע הזה (ועוד) נשמר באובייקט context. הוא מספק לנו הקשר לתהליך הבדיקה ומספק לנו מידע שנוכל להשתמש בו, לדוגמא לדעת האם הטסט עבר או נכשל, כפי שראינו במימוש שבמחלקת ההרחבה שכתבנו.

הקוד הבא לדוגמא כותב לדו"ח את שם הטסט לפני תחילת הטסט:

```
@Override
public void beforeTestExecution(ExtensionContext context) throws Exception {
  ReportManager.newTest(context.getTestMethod().get().getName());
}
```

לאחר הטסט, במידה והטסט נכשל, כותב לדו"ח שהטסט נכשל, הכל מתוך מידע שקיים ב- context:

```
@Override
public void afterTestExecution(ExtensionContext context) throws Exception {
  if (context.getExecutionException().isPresent()) {
     ReportManager.currentTest().error("An Error occured.
     Reason: " + context.getExecutionException().get().getMessage() + "  See
     logs for further details");
}
```

אובייקט context קיים בהרבה הרחבות, כולל ב- Junit 5 אך לא בכולן, ולכן לא נדון בו בהרחבה בפוסט הזה.

אתן מוזמנות לקרוא עוד על אובייקט context ולרתום אותו לצרכים שלכן.

סיכום

נגענו בנושא ההרחבות ב- Junit 5 רק על קצה המזלג. לא הזכרנו את היכולת לבצע הרחבות רק לטסט ספציפי ולא למחלקה שלמה, לא מימשנו את כל סוגי ה- extensions שהופיעו בתרשים מחזור החיים, ולא הרחבנו את נושא ה- context מעבר לבסיס. פשוט קצרה היריעה....

אם יש משהו אחד מכל הפוסט שארצה שתקחנה אתכן הלאה הוא העקרון המנחה, אותו גם ציינו בפוסט הקודם:

בצורה רחבה יותר, עקרון ההרחבה נכון לכל framework, והוא עובד בצורה דומה. לפעמים על ידי מימוש interface, לפעמים על ידי ירושה של מחלקה שמממשת את ה- listener, אבל כמעט תמיד באותו רעיון. לכן, בפעם הבאה שנראה לכן שעבור קטע קוד קטן צריך לכתוב מעטפת גדולה, והמהות הולכת לאיבוד, או שכותבים הרבה מאוד utilities, הדרך הנכונה היא לחפש את מנגנון ה- Extension הקרוב אליכן, ולרתום אותו לצרכים שלכן.

בהצלחה.