**Как в JUnit проверять ожидаемые исключения?**

[1. @Test](http://barancev.github.io/junit-catch-throwable/#toc0)

[2. try-catch](http://barancev.github.io/junit-catch-throwable/#toc1)

[3. @Rule](http://barancev.github.io/junit-catch-throwable/#toc2)

[4. AssertJ / catch-throwable](http://barancev.github.io/junit-catch-throwable/#toc3)

**Смотреть с**

[**5. JUnit 5**](http://barancev.github.io/junit-catch-throwable/#toc4)

Иногда возникновение исключения является ожидаемым поведением системы, и в тестах нужно проверять, что оно действительно возникает.

Ниже описаны пять способов, как в тестовом фреймворке JUnit перехватить ожидаемое исключение и проверить его свойства. Первые четыре из них можно использовать в JUnit 4, а последний способ использует новые возможности JUnit 5.

В качестве примера для демонстрации возьмём тест для функции стандартной библиотеки, создающей временный файл. Будем проверять, что при попытке создания файла в несуществующей директории возникает исключение типа IOException. При этом предварительно в том же самом тесте создаётся временная директория и тут же удаляется, так что мы получаем гарантированно несуществующую директорию, в которой и пытаемся создать файл:

**import** org.junit.Test**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**public** **class** **MyTest** **{**

**@Test**

**public** **void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

**Path** tmpFile **=** **Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);**

**}**

**}**

Разумеется, в таком виде тест упадёт, а в отчёте будет написано, что возникло исключение. А нам нужно, чтобы тест в этом случае наоборот помечался как успешный. Посмотрим, как это можно исправить.

**1. @Test**

Самый простой способ сообщить тестовому фреймворку о том, что ожидается исключение – указать дополнительный параметр expected в аннотации @Test:

**import** org.junit.Test**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**public** **class** **MyTest** **{**

**@Test(**expected **=** **IOException.**class**)**

**public** **void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

**Path** tmpFile **=** **Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);**

**}**

**}**

Этот параметр должен содержать тип ожидаемого исключения. Если возникнет исключение именно такого типа – тест пройдёт успешно. Если возникнет исключение другого типа или не возникнет вовсе – тест упадёт.

Достоинства:

* Простота и краткость.

Недостатки:

* Нельзя проверить текст сообщения или другие свойства возникшего исключения.
* Нельзя понять, где именно возникло исключение. В рассматриваемом примере оно могло быть выброшено не тестируемой функцией, а чуть раньше, при попытке создать временную директорию. Тест даже не смог добраться до вызова тестируемой функции – но при этом в отчёте он помечается как успешно пройденный!

Вторая из упомянутых проблем настолько ужасна, что я никому никогда не рекомендую использовать этот способ.

**2. try-catch**

Оба недостатка можно устранить, если перехватывать исключение явно при помощи конструкции try-catch:

**import** org.junit.Assert**;**

**import** org.junit.Test**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**public** **class** **MyTest** **{**

**@Test**

**public** **void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

**try** **{**

**Path** tmpFile **=** **Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);**

**Assert.**fail**(**"Expected IOException"**);**

**}** **catch** **(IOException** thrown**)** **{**

**Assert.**assertNotEquals**(**""**,** thrown**.**getMessage**());**

**}**

*// дальше идёт какой-то другой код*

*// в нём тоже может появиться неожиданный IOException*

*// если это случится -- тест упадёт*

**}**

**}**

Если исключение возникает до блока try – тест падает, мы узнаём о том, что у него возникли проблемы.

Если тестируемая функция не выбрасывает вообще никакого исключения – мы попадаем на fail() в следующей строке, тест падает.

Если она выбрасывает исключение неподходящего типа – блок catch не ловит его, тест опять таки падает.

Успешно он завершается только тогда, когда тестируемая функция выбрасывает исключение нужного типа.

Тест стал более надёжным, он больше не пропускает баги. А в блоке catch можно проверить свойства пойманного исключения.

**3. @Rule**

Однако работать с конструкцией try-catch неудобно.

Чтобы избавиться от неё, можно воспользоваться правилом ExpectedException, входящим в стандартный дистрибутив JUnit 4:

**import** org.junit.Rule**;**

**import** org.junit.Test**;**

**import** org.junit.rules.ExpectedException**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**import** static org**.**hamcrest**.**CoreMatchers**.**equalTo**;**

**import** static org**.**hamcrest**.**CoreMatchers**.**not**;**

**public** **class** **MyTest** **{**

**@Rule**

**public** **ExpectedException** thrown **=** **ExpectedException.**none**();**

**@Test**

**public** **void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

thrown**.**expect**(IOException.**class**);**

thrown**.**expectMessage**(**not**(**equalTo**(**""**)));**

**Path** tmpFile **=** **Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);**

thrown **=** **ExpectedException.**none**();**

*// дальше идёт какой-то другой код*

*// в нём тоже может появиться неожиданный IOException*

*// если это случится -- тест упадёт*

**}**

**}**

Теперь код имеет простую плоскую структуру, хотя общее количество строк кода, к сожалению, увеличилось.

Но главная проблема этого способа заключается в том, что проверки в таком стиле выглядят противоестественно – сначала описывается поведение, а потом вызывается функция. Конечно, это дело вкуса, но мне нравится, когда проверки располагаются после вызова тестируемой функции.

**4. AssertJ / catch-throwable**

Более красивый способ, использующий возможности Java 8, предлагают дополнительные библиотеки, такие как [AssertJ](http://joel-costigliola.github.io/assertj/) или [catch-throwable](https://github.com/Codearte/catch-exception/tree/2.x). Вот пример работы с AssertJ:

**import** org.junit.Test**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**import** static org**.**assertj**.**core**.**api**.**Assertions**.**assertThat**;**

**import** static org**.**assertj**.**core**.**api**.**Assertions**.**catchThrowable**;**

**public** **class** **MyTest** **{**

**@Test**

**public** **void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

**Throwable** thrown **=** catchThrowable**(()** **->** **{**

**Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);**

**});**

assertThat**(**thrown**).**isInstanceOf**(IOException.**class**);**

assertThat**(**thrown**.**getMessage**()).**isNotBlank**();**

*// дальше идёт какой-то другой код*

*// в нём тоже может появиться неожиданный IOException*

*// если это случится -- тест упадёт*

**}**

**}**

Обращение к тестирумой функции оформлено в виде лямбда-выражения (анонимной функции), которое передаётся в “ловушку” для исключений catchThrowable. Она перехватывает возникающее исключение и возвращает его как результат своей работы, давая возможность сохранить его в переменную и затем проверить его свойства. При этом проверки находятся после вызова тестируемой функции, читать код легче.А если исключение не возникнет – “ловушка” сама выбросит исключение и тест упадёт.

**5. JUnit 5**

Но почему нужно использовать какие-то дополнительные библиотеки, почему тестовые фреймворки сами не предоставляют удобных возможностей для работы с ожидаемыми исключениями?

Уже предоставляют. Перехват исключений в [JUnit 5](http://junit.org/junit5/) выглядит очень похоже на предыдущий пример:

**import** org.junit.jupiter.api.Test**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**import** static org**.**junit**.**jupiter**.**api**.**Assertions**.**assertNotNull**;**

**import** static org**.**junit**.**jupiter**.**api**.**Assertions**.**assertThrows**;**

**public** **class** **MyTest** **{**

**@Test**

**void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

**Throwable** thrown **=** assertThrows**(IOException.**class**,** **()** **->** **{**

**Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);** **});**

**//а это дополнительная проверка возвращаемого исключения**

assertNotNull**(**thrown**.**getMessage**());**

*// дальше идёт какой-то другой код*

*// в нём тоже может появиться неожиданный IOException*

*// если это случится -- тест упадёт*

**}**

**}**

**Еще один пример:**

**Метод для тестирования:**

public class Calculator {

public double divide(double a, double b) {

if (b == 0.0) {throw new IllegalArgumentException("Divider must not be 0");}

return a/b;

}}

**Тест:**

public class CalculatorTest {

@Test

void triangularMinus5() { // в JUnit5 метод тестирования не должен быть публичным

Calculator calc = new Calculator();

IllegalArgumentException thrown = assertThrows(

IllegalArgumentException.class,

() -> calculator.divide(42.0, 0.0));

// Если исключение не было вброшено, вышеуказанный тест не пройден.

// И теперь вы можете дополнительно проверить возвращенное исключение ...

// ... например как это:

assertEquals("Divider must not be 0", thrown.getMessage());

}

Раньше такая возможность в JUnit отсутствовала, потому что предыдущие версии JUnit были ориентированы на более старые версии Java, где не было лямбда-выражений и написать подобный код было просто невозможно. Да, можно сделать нечто подобное с помощью анонимных классов, но это выглядит настолько ужасно, что конструкция try-catch кажется верхом изящества.

Так что если вам приходится писать тесты, в которых проверяется возникновение исключений – есть повод присмотреться к новым возможностям JUnit 5.