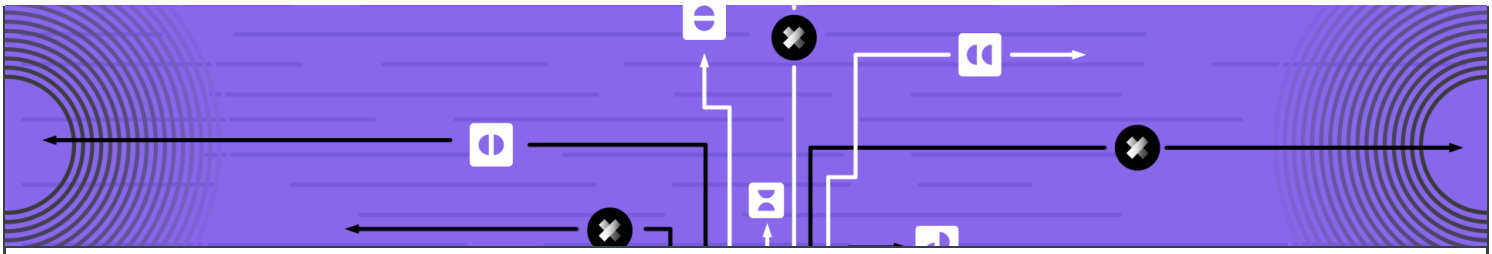

AUGUST 13TH, 2019 | 4 MINUTE READ

Chasing Swallowed Exceptions

Handling uncaught errors and crashes in testing

Tests are an essential part of every software project. They give developers confidence that any change they make doesn't break existing functionality. A test is either successful if the assumptions pass the requirements, or unsuccessful if a verification fails or the code crashes. To verify code we often rely on assertion frameworks, which not only compare the actual value with the expected one, but also print better failure messages when something goes wrong. In case of a crash we assume that our testing framework catches the exception and marks the test as failure without tearing down the whole test suite.

For the Java Virtual Machine such a crash written in Kotlin with JUnit could look like this:

```
@Test fun `let it crash`() {
 throw IllegalStateException("I did not expect this")
}
```

All JUnit does as a test framework is wrap the method call to the test function in a try-catch block. If an exception was thrown, then the test failed.

Our Point of Sale for Android codebase relies heavily on RxJava 2, a library to compose asynchronous and event-based programs using observable sequences. The codebase is growing every day and after a while we noticed suspicious stacktraces in the error log for several tests. The stacktraces looked like normal crashes as one would expect using RxJava, but the tests were successful nonetheless.

After looking into concrete scenarios and existing reported issues we realized how easy it is to reproduce these swallowed exceptions:

```
@Test fun `successful test`() {
  // Under the hood, fail() just throws an AssertionError()
  Observable.just("item").subscribe { fail("I did not expect this") }
}
```

This test is successful because of a chain of several operations. First of all, streams in RxJava 2 finish either with a complete or failure event. It's necessary to handle errors, otherwise they're forwarded to RxJava's default error handler. The test above does not have an error handler, thus the default error handler gets called. Without any configuration RxJava 2 by default does not set an error handler and calls the thread's uncaught exception handler.

That's where we enter the default JVM code path. In Java, every thread has an uncaught exception handler, which receives a callback when the thread is about to terminate due to an uncaught exception. Additionally, there is a single global handler for all threads. By default a thread's uncaught exception handler isn't set and the thread lets its thread group handle the exception. The thread group does one last check, where it looks for the global uncaught exception handler, which isn't configured either. In this case the thread group only prints the exception in the console and continues executing the code without any interruption.

That is exactly what we were observing in our test suite. Tests that should have been failing were successful, but we were seeing logged errors in the console. Because of RxJava's error handling routine, the testing framework JUnit couldn't know that there was an uncaught exception. In fact, this is not only tied to RxJava 2, it could happen with any other framework that handles exceptions and decides to call the uncaught exception handler directly.

In order to fix the issue it was clear to us that we needed our own global uncaught exception handler, which is called last in the chain of events. There we'd rethrow the exception and let JUnit handle it. An implementation could be as tiny as this:

```
internal object RethrowingExceptionHandler : UncaughtExceptionHandler {
 override fun uncaughtException(
   thread: Thread,
   throwable: Throwable
 ): Nothing = throw UncaughtException(throwable)
}

internal class UncaughtException(cause: Throwable) : Exception(cause)
```

The last question was how and when do we install the uncaught exception handler? This has to happen before any test runs. JUnit provides test rules for executing code before any test or test class starts, but this would require adding this rule to all of our test classes. Other people suggested having a base class with the rule applied and all tests to extend this class. But again, then all of our tests would need to be updated and we'd need to make sure that this is the case moving forward for new tests. With a codebase this size both solutions aren't ideal. JUnit has another mechanism: `RunListeners`. To our disappointment Gradle, the tool we use to build our code, doesn't support this infrastructure.

In the end we decided to go with a Java agent. The Java agent API was introduced with Java 5 - almost 15 years ago. A Java agent is shipped in a separate `.jar` file and has a `premain()` function, which gets called before `main()` as the name implies. This allows a Java agent to do interesting things like code instrumentation

and manipulating classes at runtime. We didn't care about any of these features, we were only interested in using the `premain()` function as a hook to install an uncaught exception handler:

```
internal object Agent {
 @JvmStatic fun premain(
   agentArgs: String?,
   instrumentation: Instrumentation
 ) {
   Thread.setDefaultUncaughtExceptionHandler(RethrowingExceptionHandler)
 }
}
```

The integration with Gradle could look like this:

```
subprojects {
 afterEvaluate { Project subproject ->
   subproject.tasks.withType(Test) {
     doFirst {
       jvmArgs "-javaagent:$javaAgentJar"
     }
   }
 }
}
```

With this setup the previously swallowed exceptions were rethrown by our uncaught exception handler and tests started to fail as we'd have hoped from the beginning.

AUTHORED BY

Ralf Wondratschek

TAGS

Android

Discuss on Twitter

Join our Slack Channel

APIs

Engineering

Data Science

View More Articles ›

Archive
Privacy Policy
Opt-Out Of Interest-Based Advertising
Slack
Twitter
Developer Support
Documentation
Work With Us