JUnit Concepts

Junit 5

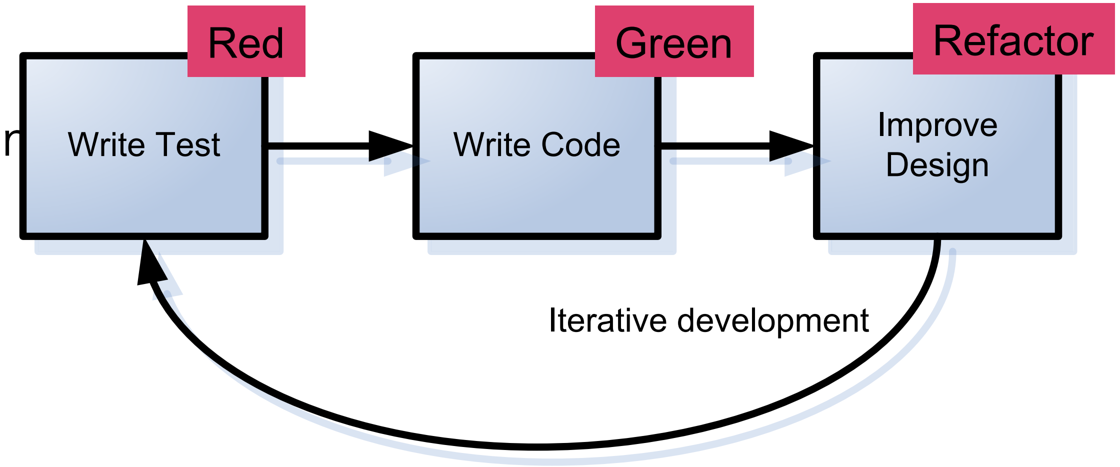
Course Objectives

In this course, we will:

- Considering a TDD approach
- How to use JUnit
- Writing test logic
- Types of assertions
- Using Test Fixtures
- Additional JUnit concepts

Steps in TDD

- Use these steps as a **rhythm** for code developm



1. Write a test
   - Nails down "public face" of the class
   - Class, library tends to be easy to use
   - At this point, the test fails ("red")
2. Implement code
   - Start off with a simple internal design for the code
   - Simplest possible implementation to get the test to pass ("green")
3. Improve design without introducing new behavior
   - Called refactoring
   - Make sure that tests continue to pass, so no new bugs are introduced

Advantages of TDD

- The first step of TDD is to write a test
  - Assuming ideal implementation of class
  - The class to be implemented tends to have a simple, easy-to-use API
    - Because programmer has not considered the implementation yet
  - Software is easy to use
- Interface of a class not closely tied to the implementation
  - Class interface tied to way client code interacts with it
  - Less likely to break because of bad assumptions
  - Software is more robust to changes
- TDD involves creating a battery of small, automated tests
  - Changes can be easily regression-tested
  - Gives programmers the confidence to make changes
  - Software can remain clean and well-designed longer

Advantages of TDD (Continued)

- The only code in the system was required by a test
  - Therefore, all code is subject to tests
    - Aim for 100% test coverage
    - Test coverage in the range of 85% is typical of Java and Java EE projects
  - Software is not bloated
    - Every bit of software is typically required by some client code somewhere
- No “marathon coding” followed by a “code freeze” followed by testing
  - Debugging tends to be faster
  - A test fails mostly because of recently added functionality
  - Since there’s a test for every branch of code, easy to find problem

TDD and Iterative Development

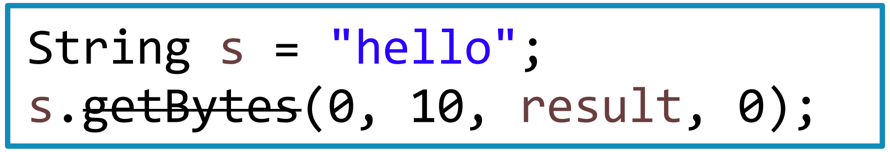
- TDD is an agile methodology
  - Involves building up a system incrementally
  - Each step takes system closer to desired end-state
  - Each step involves a test-code-refactor cycle
    - Usually under an hour
- Customers always have a working system even if it is not feature complete
  - Important to have deployable system with subset of features as soon as possible

What Is JUnit?

- JUnit is a framework to aid unit testing
  - Developed by Erich Gamma and Kent Beck
  - Forms the basis of unit testing frameworks in many other languages
  - A set of classes (UI elements also built to add support in Eclipse)
    - Simply add appropriate jar to classpath of application or as a dependency in Maven
- Many consider unit tests to be the single most important testing tool
  - Checks a single method or a set of cooperating methods
  - Tests in isolation, not the complete program
  - For each test, you provide a simple class that provides
    - Parameters to the methods being tested
    - Expected results from those methods

Java Annotations

- Java provides **annotations**
  - Can be used by Java code or external tools to mark code to act upon
  - Meta information about the code
  - Annotations start with `@`
- For example, Java uses annotation `@Deprecated` to marks a method as obsolete
  - Method still runs when called
  - But Eclipse strikes through it as a hint to developers



```
String s = "hello";
s.~~getBytes~~(0, 10, result, 0);
```

- `@Test` annotation signals to JUnit that this method should be run as a test
  - Historically, methods also tend to start with the word “test” (but do not have to)

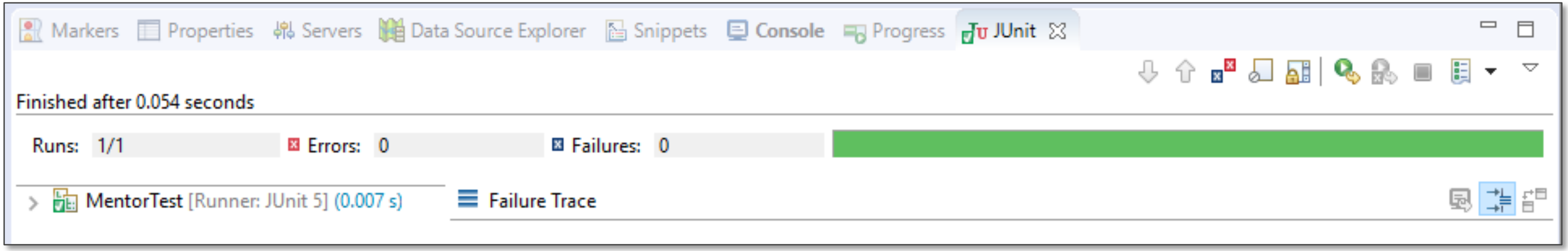
Writing Specific Tests

- Best practice to write a separate test class for each class to be tested
  - The test class is a plain Java object with methods that have the `@Test` annotation
  - This test class is called a Test Case
  - Put the Test Case in the same package as the class being tested
- Best practice for each method in Test Case to apply just one test
  - Use long names that describe what the method is testing
  - Actual test is an assertion that result should match expected value
  - A test method with no assertion passes by default (bad practice!)

```java
@Test
void testFullName() {
    String expected = "Jane Doe";
    Mentor mentor = new Mentor("Jane", "Doe");
    String actual = mentor.getFullName();
    assertEquals(expected, actual, "Full name should be Jane Doe");
}
```

Running JUnit from Eclipse

- Add build path dependency (use one of these)
  - Have Eclipse automatically add appropriate JUnit version
  - **Maven | Add Dependency** or edit pom.xml
- Right-click a class to test
  - **New | Other | Java | JUnit Test Case**
  - If using Maven, put the test case in `src/test/java`
- Right-click test class you have made
  - **Run As | JUnit Test**



Finished after 0.054 seconds

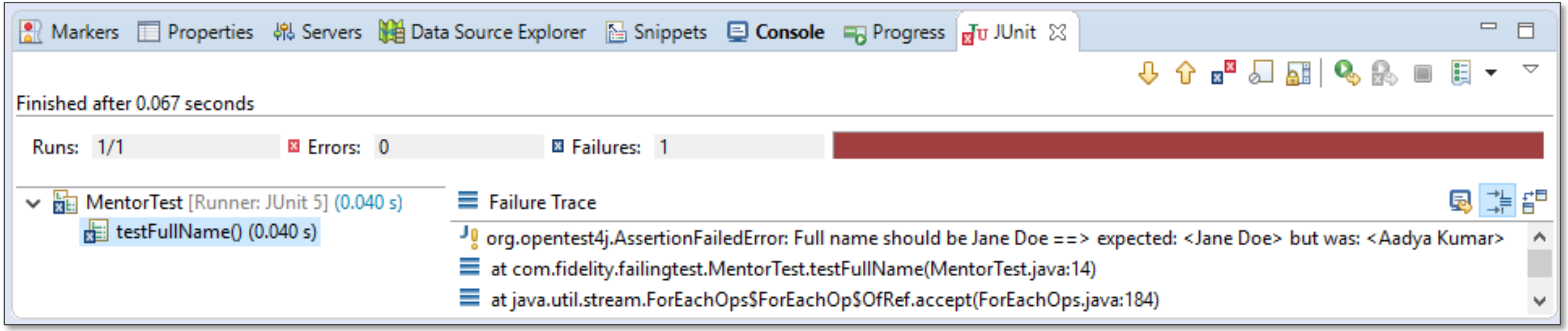
Runs: 1/1 Errors: 0 Failures: 0

MentorTest [Runner: JUnit 5] (0.007 s)

Failure Trace

Other Useful Eclipse JUnit Features

- Re-run the JUnit test
  - Click the green play in the JUnit menu or right-click a specific test



Finished after 0.067 seconds

Runs: 1/1 Errors: 0 Failures: 1

MentorTest [Runner: JUnit 5] (0.040 s)
testFullName() (0.040 s)

```
org.opentest4j.AssertionFailedError: Full name should be Jane Doe ==> expected: <Jane Doe> but was: <Aadya Kumar>
at com.fidelity.failingtest.MentorTest.testFullName(MentorTest.java:14)
at java.util.stream.ForEachOps$ForEachOp$OfRef.accept(ForEachOps.java:184)
```

- Can provide messages that display in the Runner
  - Click directly on tests to see where they failed
- Right-click project or package to re-run all tests in that scope

Testing Strategies

- Discuss what might go wrong when introducing new feature
  - This helps generate tests and especially inputs later
  - Include positive and negative (error producing) tests
- Every feature or found bug should have tests
  - It is your responsibility as a professional
- Do not test blindly: every test should have a clear purpose
- Design your code to be as easily testable as possible
- Divide what needs to be tested into different cases or categories
  - Look especially at the boundaries between cases
- Test the business logic: especially conditionals and calculations
- Do not test methods that are too simple to break (like trivial getters and setters)

Overall Steps in Writing a Unit Test

- The overall steps in testing involve:
  - Prepare test data
    - Later, we will see how Test Doubles or Mocks can help with this
  - Perform operations with system under test
  - Assert state
    - Or use Test Doubles to validate behavior
  - Destroy test data (and Mocks)
- Preparing could be:
  - Creating necessary objects
  - Connecting to resources
  - Creating files or database tables
- Destroying could be:
  - Closing files or database connections

How Do You Check the Output Is Correct?

- Calculate correct values by hand
  - E.g., for a payroll program, compute taxes manually
- Supply test inputs that provide simple ways to get the answer
  - E.g., square root of 4 is 2 and of 100 is 10
- Verify that the output values fulfill certain properties
  - E.g., square root squared = original value
- Use a simple algorithm: slow but reliable method to compute a result for testing purposes
  - E.g., use Math.pow to calculate $x^{1/2}$
  - Do **not** simply rewrite the code

Assertions

- It is a good practice to describe what you are testing in your assertion
  - `assertNotNull(Object actual, String message)`
  - `assertTrue(boolean condition, String message)`
  - `assertFalse(boolean condition, String message)`
  - `assertEquals(Object expected, Object actual, String message)`
  - `assertEquals(double expected, double actual, double delta, String message)`

Expected value

Computed value

Tolerance allowed

This message will be part of JUnit report if this test fails

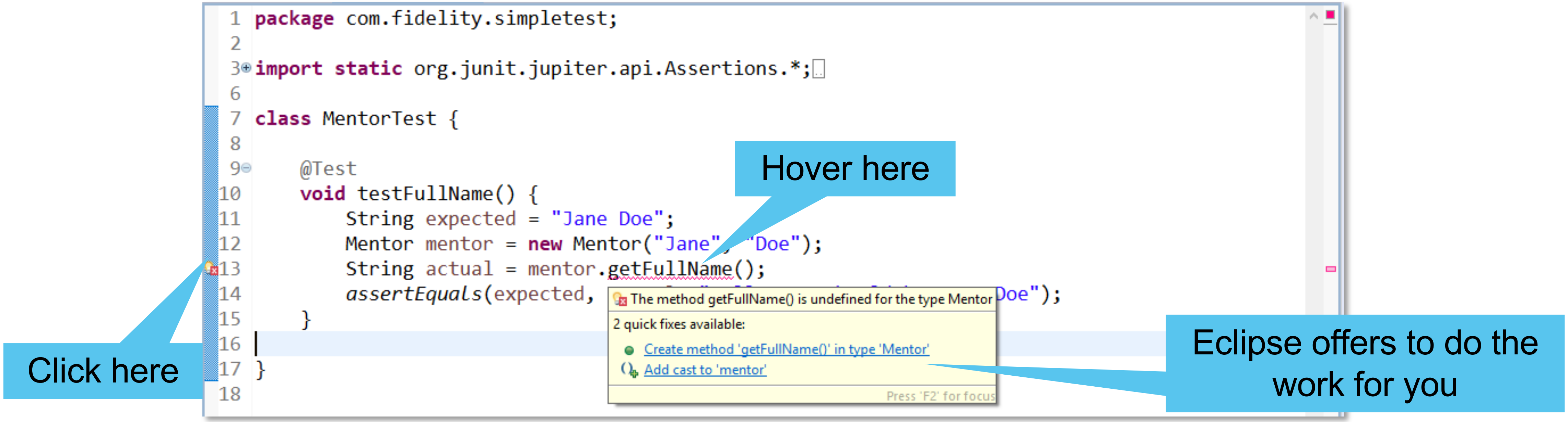
- Assertions are typically imported statically so you do not have to type the class `Assertions`

```
import static org.junit.jupiter.api.Assertions.*
```

- API documentation: https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html

Write Test Logic First

- By writing the test before implementing, we get red error squiggles



```
package com.fidelity.simpletest;

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

class MentorTest {

    @Test
    void testFullName() {
        String expected = "Jane Doe";
        Mentor mentor = new Mentor("Jane", "Doe");
        String actual = mentor.getFullName();
        assertEquals(expected,                              Doe");
    }

}

```

The method getFullName() is undefined for the type Mentor

2 quick fixes available:

- Create method 'getFullName()' in type 'Mentor'
- Add cast to 'mentor'

Press 'F2' for focus

Hover here

Click here

Eclipse offers to do the work for you

- But, more importantly, writing the tests first is part of the design process
  - What parameters should be passed? What returned?

Positive vs. Negative Testing

- Developers find it easy to produce positive tests, since that describes what they want the code to do
  - However, that ignores how the system behaves under illegal input, which is just as important
- Positive testing is the type of testing that can be performed on the system by providing the **valid data as input**
- Negative testing is a variant of testing that can be performed on the system by providing **invalid data as input**
  - For example:
    - Testing for 0 (zero) when using as a divisor
    - Testing for negative numbers when calculating a square root
    - Testing for empty string when a value is required

Exercise: Practicing TDD

- Start by following the instructor
  - Then complete this exercise described in the Exercise Manual
- Use the TDD rhythm
  - Write the test first; run it and make sure it fails (red)
  - Write only write enough code to make test pass, no more
  - Run the test again (green)
  - Repeat
- Use Eclipse—see what code Eclipse can generate for you
- Use your brain—think about what kinds of tests to make to check each line you wrote

Don’t Repeat Yourself

- Currently, we may be creating the same instances for each test
    - Just because we are writing tests is no reason to abandon good coding practices
- Objects that are created to run tests against are called a **Test Fixture**

```
@Test
public void testSimple() {
    EmailGenerator g = new EmailGenerator();
    assertEquals("doe.jane@fidelity.com",
                g.makeEmailFromName("Jane Doe"));
}

@Test
public void testExtraMiddleSpaces() {
    EmailGenerator g = new EmailGenerator();
    assertEquals("doe.jane@fidelity.com",
                g.makeEmailFromName("Jane   Doe"));
}
```

Test Fixtures: `@BeforeEach` and `@AfterEach`

- Fields are shared between tests
  - Where to set up?
- Mark initialization method with `@BeforeEach` to run before each test to create common objects or simple resources
- Mark disposal method with `@AfterEach` to run after each test completes to close resources
- Historically, these methods are named `setUp()` and `tearDown()`

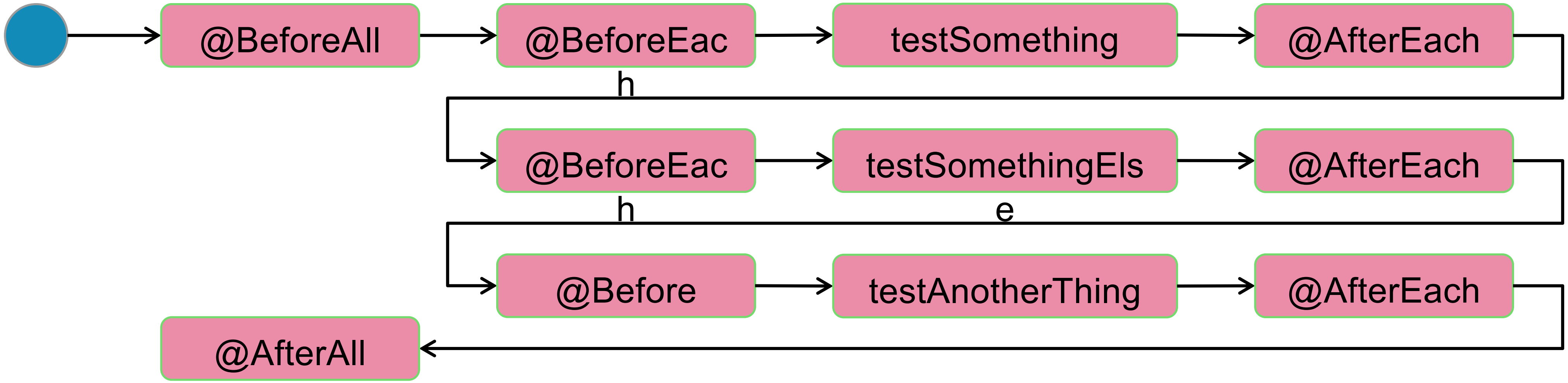
```java
private EmailGenerator g;

@BeforeEach
public void setUp() {
    g = new EmailGenerator();
}

@AfterEach
public void tearDown() {
    g = null; // Not needed in this case
}
@Test
public void testSimple() {
    assertEquals("doe.jane@fidelity.com",
                g.makeEmailFromName("Jane Doe"));
}
```

Class-Wide Setup

- Sometimes, need to do a common setup **once** for **all** tests
- Mark class-wide initialization method `@BeforeAll` to run once before any tests are run to open expensive resources like database connections
- Mark class-wide disposal method `@AfterAll` to run once after all tests have completed to close resources

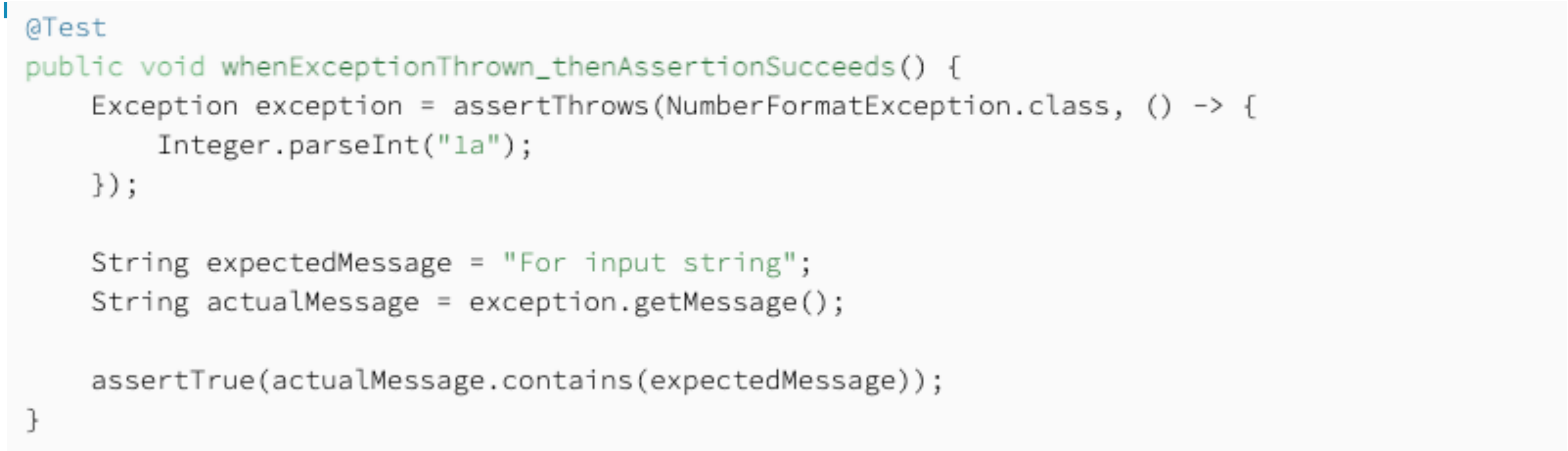
Order of Testing Methods

- In order that different tests do not interfere with each other:
  - `@BeforeAll` and `@AfterAll` methods are each called **once**
  - `@BeforeEach` and `@AfterEach` methods are called before and after **every** test
- Order in which tests are run is **not** guaranteed



Testing Exceptions in JUnit

- JUnit 5 Jupiter assertions API introduces the *assertThrows* method for asserting exceptions.
- This takes the type of the expected exception and an *Executable* functional interface where we can pass the code under test through a lambda expression:



```
@Test
public void whenExceptionThrown_thenAssertionSucceeds() {
    Exception exception = assertThrows(NumberFormatException.class, () -> {
        Integer.parseInt("1a");
    });

    String expectedMessage = "For input string";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

Why Test for Exceptions?

- Very important that error-handling receives appropriate testing
  - Many regression bugs are because of change in the way errors are handled or reported
    - Client code may expect certain behavior on certain types of inputs
    - Can not change that behavior willy-nilly
  - Important to use tests to specify error handling
  - Use tests to maintain backward compatibility of error handling

Ignore a Test

- JUnit @Disabled annotation can be used to disable the test methods from test suite.
  - This annotation can be applied over a test class as well as over individual test methods.
  - When @Disabled is applied over test class, **all test methods within that class are automatically disabled** as well.

```
public class AppTest {

    @Disabled("Do not run in lower environment")
    @Test
    void testOnDev()
    {
        System.setProperty("ENV", "DEV");
        Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));
    }
}
```

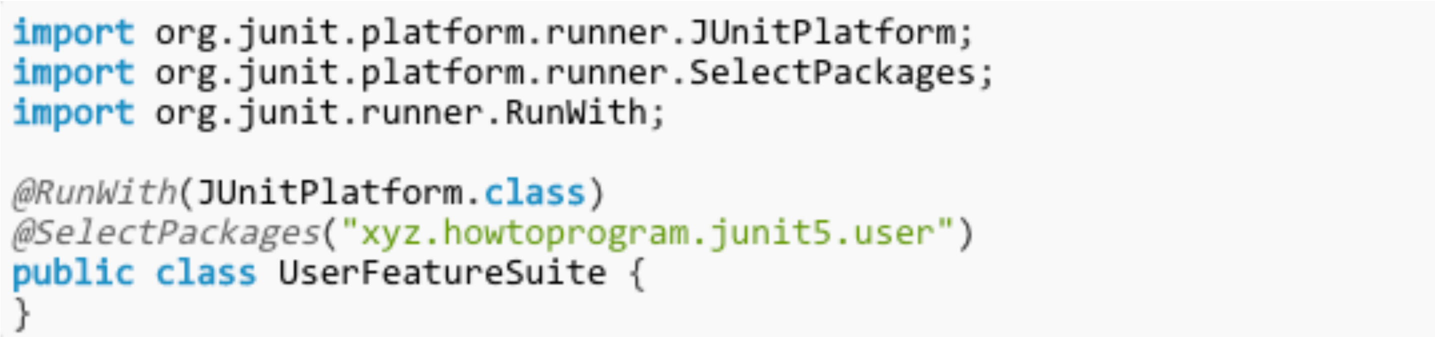
Timeout Assertion

- If a test does not complete execution in given time limit then it’s execution will be stopped by Junit.
  - In JUnit 5, we can force timeout of tests using assertions.

```
@Test
void timeoutNotExceeded()
{
    //The following assertion succeeds.
    assertTimeout(ofMinutes(2), () -> {
        // Perform task that takes less than 2 minutes.
    });
}

@Test
void timeoutExceeded()
{
    // The following assertion fails with an error message similar to:
    // execution exceeded timeout of 10 ms by 91 ms
    assertTimeout(ofMillis(10), () -> {
        // Simulate task that takes more than 10 ms.
        Thread.sleep(100);
    });
}
```

Test Suite

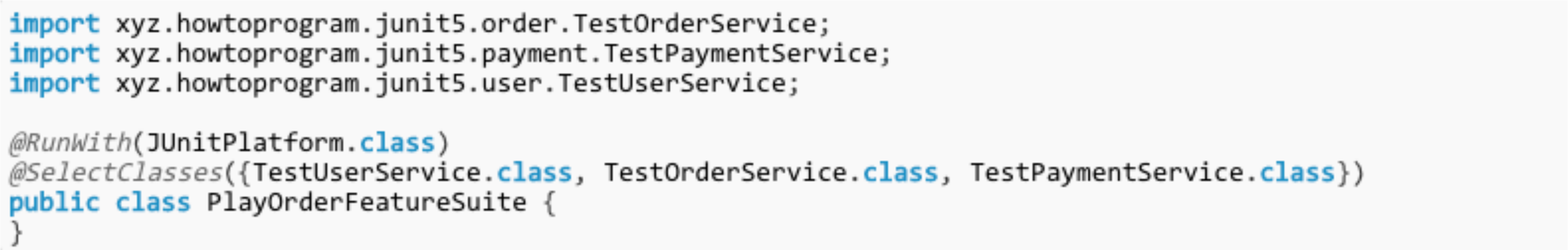
- JUnit 5 provides us 2 annotations: **@SelectPackages** and **@SelectClasses** to create test suites.
- **@SelectPackage** is used to specify the names of packages to be selected when running a test suite.



```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.runner.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("xyz.howtoprogram.junit5.user")
public class UserFeatureSuite {
}
```

Test Suite

- **@SelectClasses** is used to specify the classes to be selected when running a test suite.



```java
import xyz.howtoprogram.junit5.order.TestOrderService;
import xyz.howtoprogram.junit5.payment.TestPaymentService;
import xyz.howtoprogram.junit5.user.TestUserService;

@RunWith(JUnitPlatform.class)
@SelectClasses({TestUserService.class, TestOrderService.class, TestPaymentService.class})
public class PlayOrderFeatureSuite {
}
```