EECS2030 (B) Advanced Object-Oriented Programming Fall 2026

Digest of Lecture 10, October 19

Exception Expectations and Multi-Phase JUnit Tracing

Jackie Wang

How to use this Digest

Use this digest as a **roadmap and checklist**. It highlights the main ideas, important examples, common pitfalls, practice targets, and where to return in the course materials. It does **not** replace attending or reviewing the session, using the recording when needed, studying Jackie's annotated iPad notes and other official course materials, completing assigned work, or practising independently. Use it to identify what you understand, what you need to revisit, and what you should be able to do on your own.

If time is limited, start with the sections most relevant to your current review and return to the rest later; selective use is better than skipping the digest because it looks long.

This is an AI-assisted study roadmap reviewed by Jackie and grounded in the session transcript and Jackie's annotated iPad notes. Like any course document, it may still contain an error or omission. If something seems technically doubtful or inconsistent with the original course materials, return to those materials and ask Jackie for clarification rather than treating the digest as final authority.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 8

Where to Review 8

Closing Reminder 9

Main Ideas

- **Start a JUnit exception test from its expectation.** Jackie organized the session around two basic expectations: a particular exception *should occur*, or it *should not occur*. The test should pass only when the implementation's runtime behaviour matches that expectation.
- **The placement of `fail(...)` follows execution flow.** If an exception is *not* expected, put the failure in the matching `catch`: reaching that catch means the unwanted exception occurred. If an exception *is* expected, put the failure immediately after the method call inside the `try`: reaching that line means the expected exception did not occur.
- **A thrown exception bypasses the rest of the current `try` block.** When the method under test throws, normal execution is disrupted and control transfers to the matching `catch`. Students should trace this change in control flow rather than reading the code only top-to-bottom.
- **A good test distinguishes correct from incorrect implementations.** Jackie repeatedly ran the same JUnit test against a correct implementation and an intentionally incorrect boundary condition. A useful test passes the correct version and fails the incorrect version for the reason predicted by the test's expectation.
- **One test can contain phases with different expectations.** The third example first increments from 0 to the maximum without expecting `ValueTooLargeException`, then performs one more increment and expects that exception. The test passes only if both phases satisfy their own expectation.
- **Fail as soon as the first error is established.** In the multi-phase example, an error in Phase 1 causes immediate failure; Phase 2 is not reached. Jackie emphasized this when asking students to review the seven traced scenarios before Wednesday.

Roadmap

1. Immediate priorities after reading week

Page 2 of Jackie's annotated iPad notes records the main priorities: **Programming Test 1 is Wednesday, October 21**, during the in-room lab session, and **Lab2 is due Wednesday, October 28**. Jackie recommended using the released Programming Test 1 guide and practice test to practise coding and identify gaps. He also said Lab2 will be the basis for Programming Test 2.

Programming Test 0 results have been released. The automated feedback is intentionally limited to items such as JUnit pass/fail information; students who want deeper feedback on their submitted code can bring it to Jackie during an office-hour discussion.

2. The mental model: implementation, expectation, pass or fail

Page 3 gives the conceptual picture used throughout the lecture. Treat the **method under test** as an implementation that may be correct or incorrect. The JUnit test has a specific

expectation, such as `ValueTooLargeException` occurring or not occurring. The test's job is to produce the appropriate pass/fail result after observing the implementation.

Jackie emphasized the tester/implementer distinction and introduced a black-box view: when designing the test, begin with the intended behaviour rather than assuming how one candidate implementation works internally.

3. Test 1: `ValueTooLargeException` is not expected

Pages 4–6 develop the first test. The counter starts at 0 and `c.increment()` should move it to 1. Therefore, `ValueTooLargeException` is *not* expected.

The essential test structure is:

```
try {
    c.increment();
    assertEquals(1, c.getValue());
}
catch (ValueTooLargeException e) {
    fail("ValueTooLargeException is not expected.");
}
```

The placement of `fail(...)` is deliberate. If `c.increment()` completes normally, execution proceeds to the state assertion and the catch is bypassed. If the method unexpectedly throws `ValueTooLargeException`, the remainder of the `try` is bypassed, the matching catch executes, and the test fails.

4. Test 1 on correct and incorrect implementations

Page 5 shows the correct boundary rule: throw only when the current value equals `Counter.MAX_VALUE`. Starting from 0, the equality test against maximum 3 is false, so the method increments to 1 and returns normally. The JUnit assertion confirms the new value, the catch is bypassed, and the test passes.

Page 6 intentionally changes the condition so the method throws while the value is below the maximum. Starting from 0, the condition is already true. The unexpected exception disrupts the flow, the rest of the `try` is bypassed, and the matching catch reaches `fail(...)`. The same test now fails the incorrect implementation.

5. Test 2: `ValueTooSmallException` is expected

Pages 7–9 reverse the expectation. The counter starts at its minimum value 0, and `c.decrement()` attempts to move below that minimum. Therefore, `ValueTooSmallException` *is* expected.

The important structure is:

```
try {
    c.decrement();
    fail("ValueTooSmallException is expected.");
}
catch (ValueTooSmallException e) {
```

```
    // Exception is expected.
}
```

Jackie explicitly asked why the failure moved out of the catch. If the expected exception occurs, the next line in the `try` is bypassed and control enters the catch, where no failure is needed. If the exception does *not* occur, normal flow reaches the next line, which must fail the test.

6. Test 2 on correct and incorrect boundary logic

Page 8 uses the correct rule: when `value == Counter.MIN_VALUE`, `decrement()` throws `ValueTooSmallException`. At value 0, the expected exception is thrown, the failure line is bypassed, the catch is reached, and the test passes.

Page 9 changes the condition so the exception is thrown only when the value is strictly below the minimum. At 0, the condition is false; the implementation incorrectly decrements to -1, returns normally, and then executes `fail(...)`. The test fails because the expected exception did not occur.

7. Test 3: expectations change between phases

Page 10 combines the two patterns in one JUnit test.

Phase 1: starting from 0, call `increment()` three times to reach the maximum value 3. No `ValueTooLargeException` is expected during these calls. An exception here should immediately fail the test.

Phase 2: after verifying that the counter reached its maximum, call `increment()` once more. Now `ValueTooLargeException` is expected. If the method returns normally, the following `fail(...)` must execute.

Jackie stressed the sequential dependency: Phase 2 is reached only if Phase 1 has already satisfied its expectation.

8. Test 3 on the correct implementation

Page 11 traces the correct equality boundary. The first three calls move $0 \to 1 \to 2 \to 3$ without throwing. The state assertion confirms the maximum. The fourth call sees the value at the maximum and throws `ValueTooLargeException`. Because that exception is expected in Phase 2, the failure line is bypassed and the test passes.

9. Test 3: two different reasons incorrect implementations fail

Page 12 shows an implementation that throws while the current value is below the maximum. It fails during Phase 1 on the first increment: the exception is unexpected, so the Phase-1 catch immediately fails the test. Phase 2 is never reached.

Page 13 shows the opposite boundary error: throw only when the current value is strictly greater than the maximum. Phase 1 happens to reach 3 successfully. In Phase 2, the fourth call incorrectly changes the value from 3 to 4 instead of throwing. Execution then reaches the Phase-2 `fail(...)` line, so the test fails for a different reason.

10. Where Lecture 10 stops

Jackie said that by the end of the session the class had traced seven scenarios. Before Wednesday, students should go over those scenarios and make sure the expectation and execution flow of each one are clear. Wednesday will continue with additional conceptual/examples work to finish this part of the lecture. Jackie also distinguished today's simplified high-level tracing from full assessment tracing: when full tracing is requested, students should include the method entries, condition evaluations, state updates, returns, bypasses, and catch transitions.

Key Terms, Notation, and Structures

Method under test
The implementation being exercised by the JUnit test. Jackie treated it as the behaviour the tester must assess, rather than something the test should assume is correct.

Black-box testing perspective
The conceptual view Jackie used when setting up the session: design the test from externally expected behaviour without assuming that a candidate implementation is correct.

Test expectation
The behaviour the JUnit test is supposed to verify. In this lecture, the central distinction is whether a particular exception is expected to occur or expected not to occur.

`fail(...)`
A JUnit assertion that fails the test unconditionally if execution reaches that line. Its placement is determined by the expectation and the `try-catch` control flow.

Unexpected exception
An exception that contradicts the current test expectation. In Test 1, reaching the matching catch signals that an unwanted exception occurred.

Expected exception
An exception whose occurrence is the behaviour the test is trying to confirm. In Test 2, reaching the matching catch is compatible with passing.

Phase
A sequential portion of a test with its own expectation. Test 3 has a first phase in which `ValueTooLargeException` is not expected and a second phase in which it is expected.

Immediate failure principle
Once the test has established the first violation of its expectation, fail immediately rather than continuing to later phases.

Key Examples and Demonstrations

Test 1: no exception expected

Setup. The counter begins at 0. One increment should produce 1, so `ValueTooLargeException` is not expected.

Main point. Put `fail(...)` in the matching catch. Reaching the catch means the unexpected exception occurred.

What to practise. Reproduce pages 5 and 6 side by side. For each implementation, identify the first point where runtime behaviour either matches or contradicts the expectation, then follow the resulting control flow.

Test 2: exception expected

Setup. The counter is already at its minimum value 0. A decrement should throw `ValueTooSmallException`.

Main point. Put `fail(...)` immediately after the method call. If the expected exception occurs, that line is bypassed. If the method returns normally, reaching that line proves the expectation was violated.

What to practise. Compare pages 8 and 9 and explain why equality at the minimum is the correct boundary while "strictly below the minimum" is too late.

Test 3: two phases in one test

Setup. Phase 1 performs three increments from 0 to 3 with no exception expected. Phase 2 attempts one more increment with `ValueTooLargeException` expected.

Main point. The same test can legitimately use both failure patterns, but only if the expectation at each phase is explicit.

Three outcomes to reproduce.

- Correct equality boundary: Phase 1 succeeds, Phase 2 throws as expected, so the test passes.
- Throw while below the maximum: Phase 1 throws too early, so the test fails immediately and Phase 2 is never reached.
- Throw only above the maximum: Phase 1 reaches the maximum, but Phase 2 fails because the fourth increment returns normally and reaches `fail(...)`.

What to practise. Redo pages 11–13 without looking at Jackie's step numbers. Then, if full tracing is requested, add every method entry, condition evaluation, state update, return, bypass, and catch transition.

Common Pitfalls and Misconceptions

- **Placing `fail(...)` without first stating the expectation.** Jackie repeatedly said to decide whether the exception should occur before deciding where the failure belongs.

- **Putting the failure in the catch when the exception is expected.** That would fail precisely when the implementation behaves correctly. For an expected exception, the failure belongs after the method call that should have thrown.
- **Putting the failure after the method call when the exception is not expected.** Normal execution would reach the failure even though the implementation did nothing wrong. For an unexpected exception, fail in the matching catch.
- **Continuing through the rest of a `try` after an exception.** Once the method throws, the remaining statements in that `try` are bypassed and control transfers to the matching catch.
- **Reading a `throws` clause as "the exception always occurs."** Jackie explicitly distinguished the method's potential to throw from whether the exception actually occurs for the current input and implementation.
- **Using the wrong boundary operator.** The examples show both kinds of boundary error: throwing below the maximum is too early, while waiting until above the maximum or below the minimum is too late.
- **Assuming success in one phase makes the whole test pass.** Test 3 passes only when both Phase 1 and Phase 2 satisfy their own expectations.
- **Allowing a test to continue after the first established error.** Jackie emphasized that once `fail(...)` is reached, the test fails immediately and the remaining phases are bypassed.

What You Should Be Able to Do

After this session, you should be able to:

- state whether a given JUnit scenario expects a particular exception to occur or not occur;
- decide whether a test should pass or fail by comparing that expectation with the implementation's actual runtime behaviour;
- explain why `fail(...)` belongs in the catch when an exception is not expected;
- explain why `fail(...)` belongs immediately after the method call when an exception is expected;
- trace how a thrown exception bypasses the remaining statements in the `try` and transfers control to the matching catch;
- distinguish a method's `throws` declaration from an exception actually being thrown at runtime;
- trace Test 1 on both the correct equality boundary and the too-early-throwing implementation;

- trace Test 2 on both the correct minimum boundary and the implementation that waits until the value is below the minimum;
- divide Test 3 into Phase 1 and Phase 2 and state the expectation in each phase;
- explain why the two incorrect Test-3 implementations fail at different phases; and
- expand a high-level trace into a full step-by-step trace when an assessment explicitly asks for method calls, conditions, state changes, bypasses, catches, and pass/fail outcomes.

Suggested Practice

- Prepare for **Programming Test 1 on Wednesday, October 21** by working through the released guide and practice test, including practising the coding workflow under time pressure.
- Keep pace with **Lab2, due Wednesday, October 28**. Jackie warned against leaving it until the final one or two days and said it will support later Programming Test 2 preparation.
- Put Test 1 and Test 2 side by side and explain, without memorizing the code, why the `fail(...)` assertion belongs in different places.
- Reproduce the correct and incorrect traces for Test 1 and Test 2 from Jackie's annotated iPad notes. At each thrown exception, explicitly mark the remainder of the `try` as bypassed and identify the matching catch.
- Reproduce all three Test-3 traces from pages 11–13. Identify the exact point at which the test either satisfies or violates the expectation of the current phase.
- Take one of today's high-level traces and redo it at full granularity: include method entry, condition evaluation, state update, method return, assertion, bypass, catch transfer, and final pass/fail outcome.
- Before Wednesday, review the seven scenarios Jackie said had already been traced in this session. The next class will continue with more conceptual/examples work on the same material.

Where to Review

- **Jackie's annotated iPad notes, page 2:** post-reading-week priorities, Programming Test 1 on October 21, Lab2 due October 28, and the Programming Test 0 feedback reminder.
- **Jackie's annotated iPad notes, page 3:** the method-under-test / JUnit-test mental model connecting implementation, expectation, and pass/fail.
- **Jackie's annotated iPad notes, pages 4–6:** Test 1, `ValueTooLargeException` not expected, including the test design and the correct-versus-incorrect implementation traces.

- **Jackie's annotated iPad notes, pages 7–9:** Test 2, `ValueTooSmallException` expected, including why `fail(...)` moves after `c.decrement()` and the two boundary traces.
- **Jackie's annotated iPad notes, page 10:** Test 3's two-phase design: three safe increments followed by one increment at the maximum.
- **Jackie's annotated iPad notes, pages 11–13:** Test 3 on the correct implementation and on the two different incorrect boundary implementations.
- **Lecture recording, roughly 0:00–0:09:** post-reading-week logistics, tester/implementer roles, black-box perspective, expectations, and the session's JUnit testing principle.
- **Recording, roughly 0:09–0:29:** Test 1, placement of `fail(...)` for an unexpected exception, and the detailed correct and incorrect increment traces.
- **Recording, roughly 0:29–0:45:** Test 2, the changed failure placement, and the correct/incorrect decrement traces.
- **Recording, roughly 0:45–1:08:** Test 3's two-phase expectation, the correct implementation, the two distinct incorrect implementations, and Jackie's review guidance for Wednesday.

Closing Reminder

Use this digest as a roadmap for active review, not as a substitute for the learning process itself. Attend Jackie's in-person sessions and pay attention to how he decides the expectation first and then derives the JUnit control flow from that expectation. If you are unsure why a statement is bypassed, why a catch is entered, or why a particular `fail(...)` is reachable, return to the recording to fill that gap.

For structured review, start with Jackie's annotated iPad notes because they most closely reflect the actual flow and development of the class. Work through them in manageable logical units. After each logical unit, study the corresponding part of the presentation slides alongside it. Jackie does not normally teach by reading the slides line by line, so the slides may contain relevant supporting material that was not explicitly read aloud.

Finally, independently reproduce a suitable selection of the important reasoning from this session: trace Test 1 with an unexpected exception, trace Test 2 with an expected exception, and work through the three Test-3 implementations while tracking the expectation of each phase. The goal is not to memorize Jackie's step numbers, but to decide from the test's expectation where execution should flow and why the resulting implementation should pass or fail.