EECS2030 (B) Advanced Object-Oriented Programming Fall 2026

Digest of Lecture 8, October 5

Exception Handling Flows: Catch, Specify, and Debugging

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 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 7

Suggested Practice 8

Where to Review 8

Closing Reminder 9

Main Ideas

- **The same method chain can produce different execution flows.** Jackie compared normal and exceptional runs of `Tester.main -> B.mb -> A.ma`. When no exception is thrown, control returns to the next normal statement. When `A.ma` throws, normal continuation is interrupted and control follows the exception-handling path.
- **The first catch determines how far the exception propagates.** In Version 1, `B.mb` catches, so `Tester.main` is no longer subject to catch-or-specify for that exception. In Version 2, `B.mb` specifies and `Tester.main` catches. In Version 3, all three methods specify, so the exception reaches the console.
- **A redundant downstream catch can be rejected by the compiler.** Jackie demonstrated that once `B.mb` has already caught `NegValException`, trying to catch that same exception around the call to `B.mb` in `Tester.main` produces an unreachable-catch compilation error.
- **Paper tracing and debugger tracing should agree.** Jackie first labelled the execution path on paper, then used Eclipse breakpoints, `Step Into`, and `Step Over` to verify the same path for positive and negative inputs.
- **Catch versus specify changes what happens after a throw.** If a method catches, execution continues through its catch-handling path. If it specifies, its normal continuation is bypassed and the exception moves to its caller.
- **The catch-or-specify model generalizes to an arbitrary call stack.** Under the lecture's convention that the origin method specifies, if the first catch is at `Ci.mi`, then the methods from `C1.m1` through `Ci.mi` are the ones subject to the requirement: the first $i-1$ specify and the i-th catches.

Roadmap

1. Immediate priorities

Lab1 is due **Wednesday, October 7 at 11:59 p.m.** Jackie clarified that the deadline is after the lab session; TAs will be available on Wednesday for Lab1 questions. He also said Programming Test 0 grading would begin during the week, with a best-effort goal of returning results and feedback by the end of reading week.

For this lecture's technical material, Jackie asked students to revisit the exception traces on paper and in Eclipse before Wednesday.

2. Recap the catch-or-specify model before tracing

The lecture starts by returning to the call-stack model from Lecture 7. When an exception originates at the active method on top of the stack, Jackie's course convention is that the origin method *typically specifies* and propagates rather than catching locally. Each later caller can either specify again or catch.

The decisive point is the **first catch**. Once a method catches that exception, callers below it in the stack are no longer subject to the catch-or-specify requirement for that exception. Jackie also explained that the choice of where to place that first catch is a software-design decision: a method may propagate because it lacks the context or responsibility needed to handle the problem appropriately.

3. Version 1: `B.mb` catches

Page 3 of Jackie's annotated iPad notes shows Version 1 with the call chain:

```
Tester.main -> B.mb -> A.ma
```

The custom exception shown there is `NegValException`.

A negative input makes `A.ma` follow the abnormal branch and throw the custom exception. Its header specifies that exception. Caller `B.mb` puts the call in a `try` block and catches it locally.

Because `B.mb` catches, the exception is not propagated to `Tester.main`. Jackie verified this in Eclipse by temporarily adding a catch for `NegValException` around `B.mb` in `Tester.main`. Eclipse reported an unreachable catch block because `B.mb` does not expose that exception to its caller in this version.

Jackie's practical check was to inspect the callee's header: if the called method does not specify that exception in its header, the caller is not expected to catch it in this example.

4. Version 1 normal flow: positive input

The green path on page 3 uses a positive input, shown as 5. The call chain is built normally:

$$\texttt{Tester.main} \rightarrow \texttt{B.mb} \rightarrow \texttt{A.ma}.$$

Inside `A.ma`, the negative-input condition is false, so the normal branch executes. `A.ma` terminates normally, `B.mb` resumes after the call, the catch block is bypassed, and `Tester.main` later resumes after its call to `B.mb`.

Page 4 isolates the key control-flow idea: if the call inside the `try` completes without throwing, execution goes to the next normal statement and skips the associated `catch`.

5. Version 1 abnormal flow: negative input

The red/pink path on page 3 uses a negative input, shown as -4. The beginning of the trace is the same until `A.ma` evaluates the condition. This time the abnormal branch executes and the exception is thrown.

At that point, `A.ma`'s normal continuation stops. Control returns toward `B.mb`, but because an exception is being propagated, the statement after the call in the `try` is bypassed. Control transfers to the matching `catch`. After the catch finishes, `B.mb` terminates and `Tester.main` resumes after its call.

This contrast between the positive and negative runs is the central tracing exercise of the session.

6. Verify Version 1 with the Eclipse debugger

Jackie then reproduced both traces in Eclipse. He set a breakpoint in `Tester.main`, switched to the debug perspective, and followed calls with `Step Into` and `Step Over`.

For a positive input, stepping over the call to `A.ma` returns to the next normal statement in `B.mb`. For a negative input, stepping over that same call lands in the matching catch block instead. Jackie used this comparison to connect debugger behaviour directly to the paper trace rather than treating the debugger as a separate topic.

7. Version 2: `B.mb` specifies, `Tester.main` catches

Page 5 changes only the handling location. `A.ma` still specifies at the origin. Instead of catching in `B.mb`, the method header for `B.mb` now includes:

```
throws NegValException
```

so the exception is propagated to `Tester.main`. The call in `Tester.main` is then placed inside a `try-catch` that handles `NegValException`.

For the negative-input trace, the key difference from Version 1 is that after `A.ma` throws, `B.mb` does *not* resume with its next normal statement. Since `B.mb` specifies, the exception continues to `Tester.main`, where the catch block handles it.

8. Version 3: nobody catches

Page 6 shows the deliberately undesirable extreme. `A.ma` specifies, `B.mb` specifies, and `Tester.main` also specifies. With a negative input, no catch exists anywhere in the call stack, so the exception is propagated beyond `main` to the console.

Jackie demonstrated that this version compiles, but described it as a design to avoid. The tracing also shows that the normal print statements after the exception-producing calls are bypassed; execution does not resume those normal paths while the exception is propagating.

9. Generalize to n methods

Page 7 abstracts the three-version examples into a call stack of n methods. The top method is `C1.m1`, where the exception originates; the entry point is `Cn.mn` at the bottom.

Assume the first catch occurs at `Ci.mi`. Then:

- `C1.m1` through `C(i-1).m(i-1)` specify;
- `Ci.mi` catches; and
- `C(i+1).m(i+1)` through `Cn.mn` are free from this exception's catch-or-specify requirement.

Therefore i methods are subject to the requirement in this setup: $i - 1$ specify and the i-th catches. Jackie confirmed this with an in-class choice among $i - 1$, i, and $i + 1$.

Under the lecture's origin-method convention, the earliest catch is at `C2.m2`. The opposite extreme is that nobody catches, in which case all n methods specify and the exception reaches the console.

Key Terms, Notation, and Structures

Catch-or-specify requirement
The handling discipline used throughout the session: a method receiving a propagated exception either handles it with a catch or specifies it further.

First catch
The first method in the propagation path that handles the exception. In the lecture model, propagation and catch-or-specify obligations for later callers stop at this point.

Specify
Propagate the exception to the caller using a method-header `throws` declaration.

Catch
Handle the exception locally with a matching `catch` block associated with a `try` block.

`NegValException`
The custom exception class used in Jackie's annotated iPad notes for the Version 1–3 examples.

Normal flow
The path followed when `A.ma` does not throw. The call returns normally, execution continues after the call, and the catch is bypassed.

Abnormal flow
The path followed when `A.ma` throws. Normal continuation is interrupted and control follows the exception-propagation/handling path.

Breakpoint
A debugger stop point used by Jackie to pause execution before tracing the method chain.

`Step Into` / `Step Over`
Debugger operations used to enter a called method or execute a call without manually stepping through every line of the callee.

`Ci.mi`
Jackie's generalized notation on page 7 for the method at which the first catch occurs in an n-method call stack.

Key Examples and Demonstrations

Version 1: same code, two different flows

Setup. Page 3 of Jackie's annotated iPad notes shows Version 1 with `A.ma` specifying and `B.mb` catching.

Positive input. With `i = 5`, the condition in `A.ma` is false. The normal branch executes, `A.ma` returns normally, `B.mb` continues after the call, and the catch is skipped.

Negative input. With `i = -4`, the abnormal branch executes and `A.ma` throws. The next normal statement in `B.mb`'s `try` is bypassed and control transfers to the matching catch.

What to practise. Reproduce both coloured paths from page 3 without looking at the final labels. For every numbered step, state which method is active and whether the next transition is a normal return or exception transfer.

The unreachable-catch demonstration

Setup. Version 1 already catches `NegValException` inside `B.mb`.

Main point. Jackie temporarily tried to catch the same exception again around the call to `B.mb` in `Tester.main`. Eclipse rejected the catch as unreachable because the statement body does not throw that exception in this version.

What to practise. Before writing a catch around a method call, inspect the callee's header and determine what exception, if any, is actually specified to its caller.

Debugger confirmation of the two Version 1 paths

Setup. Jackie places a breakpoint before the call from `Tester.main` to `B.mb` and traces with Eclipse.

Main point. At the call from `B.mb` to `A.ma`, `Step Over` has two visibly different outcomes:

- with positive input, it returns to the next normal statement; and
- with negative input, it transfers to the matching catch block.

What to practise. Predict the landing point before clicking the debugger control, then verify the prediction in Eclipse.

Version 2: move the catch one level down the stack

Setup. Page 5 has `A.ma` specify, `B.mb` specify, and `Tester.main` catch.

Main point. When `A.ma` throws, `B.mb`'s normal continuation does not execute. The exception passes through `B.mb` and is handled by `Tester.main`.

What to practise. Trace the negative-input path on paper, then repeat it with the debugger using the same breakpoint/stepping strategy as Version 1.

Version 3 and the generalized call stack

Setup. Page 6 sends the exception all the way to the console. Page 7 then generalizes the same idea to n methods and a first catch at `Ci.mi`.

Main point. The first catch partitions the stack: above it, methods specify; at it, the exception is caught; below it, later callers are no longer subject to this exception's catch-or-specify requirement. If no catch exists, all methods specify and the exception escapes the application.

What to practise. Recreate the page-7 partition and justify why the number of methods subject to catch-or-specify is i, not $i - 1$ or $i + 1$.

Common Pitfalls and Misconceptions

- **Trying to catch again after the exception has already been caught.** In Version 1, `B.mb` already handles `NegValException`; the additional catch in `Tester.main` is unreachable.
- **Ignoring the callee's method header.** Jackie explicitly recommended checking whether the callee specifies the exception before deciding that the caller should catch it.
- **Expecting normal continuation after a throw.** In the negative-input trace, the statement after the call inside the `try` is bypassed; execution transfers to the matching handling path.
- **Assuming Version 2 behaves like Version 1 after `A.ma` throws.** In Version 2, `B.mb` specifies rather than catches, so its next normal statement is also bypassed and the exception continues to `Tester.main`.
- **Treating debugger stepping as a substitute for paper reasoning.** Jackie used paper first and Eclipse second. Students were asked to predict the control-flow destination and then verify it with the debugger.
- **Letting every method specify all the way out of `main`.** Version 3 demonstrates that this compiles but sends the exception to the console. Jackie described this as a design to avoid.
- **Treating the origin-method convention as a Java impossibility.** Jackie said that catching at the origin is programmatically possible. The lecture's "principle zero" is the convention used for these examples, not a claim that Java forbids a local catch there.

What You Should Be Able to Do

After this session, you should be able to:

- identify the origin, callers, and first catch in the `Tester.main -> B.mb -> A.ma` examples;
- trace Version 1 for both a positive and a negative input;
- explain why the normal statement after a call executes in one Version 1 path but is bypassed in the other;
- explain why an extra catch in `Tester.main` is unreachable when `B.mb` already catches `NegValException`;
- inspect a callee's header and determine whether the caller is subject to catch-or-specify for the demonstrated exception;
- use a breakpoint, `Step Into`, and `Step Over` to verify a predicted exception flow in Eclipse;

- distinguish Version 1, which catches in `B.mb`, from Version 2, which specifies in `B.mb` and catches in `Tester.main`;
- trace Version 3 and explain why the exception reaches the console;
- explain how moving the first catch changes which methods are subject to catch-or-specify;
- reproduce the page-7 n-method generalization with first catch at `Ci.mi`;
- justify why i methods are subject to the requirement in that setup, while $n - i$ later callers are free from it; and
- identify the earliest catch allowed by the lecture's origin-method convention and the no-catch extreme.

Suggested Practice

- Complete Lab1 by **Wednesday, October 7 at 11:59 p.m.** Jackie said TAs will be available on Wednesday if you have Lab1 questions.
- Reproduce the complete Version 1 paper trace twice: once with the positive input and once with the negative input. Use page 3 of Jackie's annotated iPad notes only to check your work after you finish.
- Download the three exception-handling versions from the lecture-site source-code example and rerun the Version 1 debugger experiment. Predict each `Step Into/Step Over` destination before clicking.
- Trace Version 2 with a negative input and explicitly mark which normal statement in `B.mb` is bypassed because `B.mb` specifies rather than catches.
- Run or trace Version 3 and confirm that the exception reaches the console instead of any later normal print statement.
- Recreate the generalized page-7 call stack from memory. Mark the origin `C1.m1`, the first catch `Ci.mi`, the $i - 1$ methods that specify, and the $n - i$ later callers that are free from the requirement.
- Before Wednesday, revisit the tracing using both paper and Eclipse, as Jackie explicitly requested at the end of the lecture.

Where to Review

- **Jackie's annotated iPad notes, page 2:** Lab1 due Wednesday, October 7.
- **Jackie's annotated iPad notes, pages 3–4:** Version 1 with `A.ma` specifying and `B.mb` catching; the green normal-flow trace, red/pink exceptional-flow trace, and the simplified two-destination sketch for normal continuation versus the matching catch.

- **Jackie's annotated iPad notes, pages 5–6:** Version 2 (`B.mb` specifies, `Tester.main` catches) and Version 3 (all three methods specify, so the exception reaches the console).
- **Jackie's annotated iPad notes, page 7:** the generalized n-method call stack, first catch at `Ci.mi`, counts of methods subject/free from catch-or-specify, and the two extreme cases.
- **Lecture recording, roughly 0:00–0:15:** course reminders, catch-or-specify recap, Version 1 setup, and the unreachable-catch demonstration in `Tester.main`.
- **Recording, roughly 0:15–0:49:** Version 1 paper traces for positive/negative inputs and Eclipse verification using breakpoints, `Step Into`, and `Step Over`.
- **Recording, roughly 0:49–1:02:** Version 2 propagation to `Tester.main`'s catch, followed by Version 3 propagation to the console.
- **Recording, roughly 1:02–1:13:** generalized n-method call stack, first-catch position, method counts, and the earliest/no-catch extreme cases.

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 labels control-flow steps on paper before checking them in Eclipse. If a transition from normal execution to exception propagation, or from propagation to a catch block, remains unclear afterwards, return to the recording and replay that exact trace.

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 Version 1 for both positive and negative inputs, predict and verify the debugger's next location, trace the changed propagation path in Version 2, explain why Version 3 reaches the console, and rebuild the generalized n-method catch-or-specify diagram. The goal is to derive each transition from the method headers, active call stack, and first-catch location rather than memorizing Jackie's coloured arrows or debugger clicks.