EECS2030 (B) Advanced Object-Oriented Programming Fall 2026

Digest of Lecture 13, October 28

Overridden equals: Phases 1–2, Static/Dynamic Types, and Casting

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 7

Key Examples and Demonstrations 7

Common Pitfalls and Misconceptions 9

What You Should Be Able to Do 10

Suggested Practice 10

Where to Review 11

Closing Reminder 12

Main Ideas

- **The overridden `equals(Object obj)` method is developed phase by phase.** Lecture 13 focuses on Phase 1 and Phase 2, then builds the type-system background needed for Phase 3. The detailed Phase-3 implementation itself is deferred to the beginning of Monday's lecture.
- **Phase 1 handles identity and aliasing immediately.** The condition `if (this == obj) { return true; }` asks whether the context object and parameter store the same reference. If they do, the objects are trivially equal and the remaining phases are bypassed.
- **Phase 2 handles a null argument.** Reaching Phase 2 already implies that `this != obj`. If `obj == null`, `this` must therefore be non-null, so the method can return false immediately.
- **Checking `this == null` inside an instance method is redundant.** Jackie gave two reasons: a null context object would cause a `NullPointerException` before the method body begins, and reaching Phase 2 already implies `this != obj`. When `obj` is null, `this` cannot also be null.
- **Static type and dynamic type serve different purposes.** Static type is the declared type and never changes; it determines which method calls are allowed to compile. Dynamic type depends on the object currently referenced and may change after reassignment; it determines which method version is invoked at runtime.
- **A cast creates an expression with a different static type.** In Jackie's visual model, a cast such as `((C1) o3)` behaves like an alias to the same referenced object whose static type is `C1`. This can make a method call compile without changing `o3`'s declared type or creating a new object.
- **Short-circuit evaluation is assigned preparation, not material developed in this lecture.** The exercise is testable on Written Test 1, but Jackie left its detailed study to students.

Roadmap

1. Immediate priorities and Written Test 1 scope

Jackie opened with the following actions:

- **Lab2 is due October 28 at 11:59 p.m.**
- **Lab3 has been released.**
- The scheduled lab session has optional attendance and no scheduled test; TAs are available in the room.
- **Written Test 1 is Wednesday, November 4.**

For Written Test 1, students need a mobile phone for Duo verification and both a working eCS login and Passport York login. Jackie warned that time spent fixing login problems during the test will not be added back.

The test scope includes the material covered through this Lecture 13, the assigned short-circuit-evaluation exercise, and approximately the first 5–10 minutes of Monday's lecture when Phase 3 of the overridden `equals` method is completed. Jackie said the rest of Monday's material will be left to Written Test 2.

2. Recap: default versus overridden `equals`

Before starting the new phases, Jackie briefly revisited the previous lecture's dispatch model:

- a class that does not override `equals` uses the inherited default version from `Object`;
- a class that explicitly overrides `equals` uses that overridden version when the context object's dynamic type is that class.

Page 3 of Jackie's annotated iPad notes shows the full four-phase implementation: `PointV2.equals(Object obj)`. Lecture 13 develops the first two phases in detail and only previews the Phase-3 condition.

3. Phase 1: compare the context object and parameter references

The first phase is:

```
if (this == obj) { return true; }
```

Jackie first establishes the runtime bindings for a call such as `p1.equals(p2)`:

- `p1` is the context object, so `this` refers to the object referenced by `p1`;
- by call by value, the parameter receives the argument reference through the implicit assignment `obj = p2`.

Phase 1 then compares the two stored references. If they identify the same object, return true immediately. If they identify different objects, continue to the later phases.

4. Phase-1 trace: `p1.equals(p1)`

Page 4 gives the direct reflexive example. The context object `p1` has dynamic type `PointV2`, so the overridden `PointV2.equals` method is invoked. Call by value gives `obj = p1`. Therefore `this` and `obj` point to the same object.

The Phase-1 condition is true, the method returns true, and every later phase is bypassed.

5. Different variable names may still pass Phase 1

Jackie then used a hypothetical call `p1.equals(p4)` to emphasize aliasing. Merely seeing two different reference-variable names does not tell you whether the references differ.

If `p1` and `p4` are aliases to the same `PointV2` object, call by value makes `obj` point to that same object as well, so `this == obj` is still true. If they point to different objects, only then does the method continue beyond Phase 1.

6. Reaching Phase 2 gives you an invariant

The second phase is:

```
if (obj == null) { return false; }
```

Before evaluating it, Jackie emphasized the execution-flow fact that reaching Phase 2 means Phase 1 did not return. Therefore:

`this != obj`.

This is useful information carried forward from the earlier branch. Students should use these implicit conditions when tracing multi-phase methods.

7. Why Phase 2 returns false for a null argument

If `obj == null`, then the parameter does not reference an object. Because the method body is already executing, `this` must refer to a real context object. A non-null object cannot be equal to null, so the correct result is false.

Page 5 expands this idea with an intentionally redundant alternative that checks whether `this == null` inside the `obj == null` case. Jackie uses that alternative only to explain why the extra check is unnecessary.

8. Two reasons `this == null` is redundant inside `equals`

Jackie gave two distinct arguments.

Reason 1: method invocation. For a call such as `x.equals(y)`, if the context reference `x` were null, Java would raise a `NullPointerException` before entering the method body. Therefore, if execution has reached any line inside the instance method, `this` cannot be null.

Reason 2: prior-phase logic. Reaching Phase 2 means `this != obj`. If Phase 2 now establishes that `obj == null`, `this` cannot also be null. The earlier branch condition already rules that out.

These two arguments justify the compact Phase-2 statement rather than a larger case analysis.

9. Phase-2 trace: `p1.equals(null)`

Page 6 traces line 8 of the example. The context `p1` has dynamic type `PointV2`, so the overridden version is selected. Before the method body:

```
obj = null;
```

Phase 1 evaluates `this == obj` as false because `this` is non-null while `obj` is null. Phase 2 then sees `obj == null`, returns false, and bypasses the remaining phases.

10. Phase 3 preview: why more type background is needed

The next line of the standard pattern compares:

```
this.getClass() != obj.getClass()
```

Jackie said `getClass()` is a library method that reports the dynamic type of its context object. Rather than fully implementing Phase 3 immediately, he paused the equality trace and introduced the type concepts needed to reason about it correctly. Phase 3 itself will be completed at the beginning of Monday's lecture.

11. Static type: declared, fixed, and used for compilation

Page 7 defines a reference variable's **static type** as its declared type. Once declared, the static type does not change.

Jackie's key rule is:

> The static type determines the range of methods that can be invoked on the reference variable.

In this lecture, this is used as a compilation rule. To decide whether a method call is even legal, look at the static type of the expression on the left of the dot and the methods available from that type.

12. Inherited methods accumulate into the static-type expectation

Pages 7–8 use the hierarchy:

- `Object` provides `equals`;
- `C1` extends `Object` and declares `m1`;
- `C2` extends `Object` and declares `m2`.

The resulting expectations are:

- static type `C1`: `m1` and inherited `equals`;
- static type `C2`: `m2` and inherited `equals`;
- static type `Object`: `equals`.

Jackie then checks the nine combinations formed by context variables `o1:C1`, `o2:C2`, `o3:Object` and method names `m1`, `m2`, and `equals`. A call compiles only when the method belongs to the expectation of the context expression's static type.

13. Static type and dynamic type must be tracked separately

Dynamic type is the class of the object currently referenced at runtime. Unlike static type, it may change when the reference variable is reassigned.

Page 9 tracks `o3`:

- after `Object o3;`: static type is `Object`; no object is referenced yet, so the dynamic type is undefined/null;
- after `o3 = o1;`: static type remains `Object`; dynamic type becomes `C1`;
- after `o3 = o2;`: static type still remains `Object`; dynamic type becomes `C2`.

The static type never changes. The dynamic type follows the object currently reached by the reference arrow.

14. Why `o3.m1()` still does not compile after `o3 = o1`

This is the key separation between compile-time and runtime reasoning.

After `o3 = o1`, the dynamic type of `o3` is `C1`, and that object does have `m1`. However, the declared static type of `o3` remains `Object`. Since `Object`'s current expectation in this example contains only `equals`, the source expression `o3.m1()` does not compile.

Jackie used this mismatch to motivate type casting.

15. Type casting: create an alias expression with another static type

The expression

```
((C1) o3)
```

does not change the declared static type of `o3` and does not create a new object. In Jackie's diagram, it creates an alias expression pointing to the same object as `o3`, but the cast expression itself has static type `C1`.

Therefore:

- `((C1) o3).m1()` compiles because `C1` supports `m1`;
- `((C1) o3).m2()` does not compile because `C1` does not support `m2`.

After `o3 = o2`, the corresponding cast to `C2` allows `m2` but not `m1`. The static type of the cast expression controls the compile-time method eligibility.

16. Where Lecture 13 stops

Jackie ended by asking students to review the casting example carefully and even rewatch the final roughly ten minutes if needed. Phase 3 and Phase 4 of the `equals` implementation rely on this type reasoning.

For Written Test 1, students should study everything completed through the end of Lecture 13, complete the assigned short-circuit-evaluation exercise, and then add the brief Phase-3 development from the beginning of Monday's class.

Key Terms, Notation, and Structures

Phase 1 of overridden `equals`
`if (this == obj) { return true; }`. This handles the case where context object and parameter identify the same object.

Phase 2 of overridden `equals`
`if (obj == null) { return false; }`. Reaching this phase already implies `this != obj`.

Context object / `this`
The receiver to the left of the dot in the method call. In `p1.equals(p2)`, `p1` supplies the context object referred to by `this` inside the method.

Call-by-value parameter binding
Before the method body begins, the argument's stored value is copied into the parameter. For `p1.equals(p2)`, the relevant binding is `obj = p2`.

Static type
A reference variable's declared type. It is fixed and determines the range of method calls accepted at compile time.

Dynamic type
The type of the object currently referenced at runtime. It may change after reassignment and determines which overridden method version is invoked.

Expectation
Jackie's term for the methods available from a static type, including inherited methods. In the lecture's example, `C1` has expectation `m1` plus `equals`, while `C2` has `m2` plus `equals`.

Type cast
An expression such as `((C1) o3)`. In Jackie's visual model, it creates an alias to the same object with the cast type as that expression's static type.

`getClass()`
A library method previewed for Phase 3. Jackie described it as providing the dynamic type of its context object.

Key Examples and Demonstrations

Phase 1: reflexive equality

Setup. Page 4 traces `p1.equals(p1)`.

Main point. Dynamic type `PointV2` selects the overridden method. Call by value gives `obj = p1`, so `this` and `obj` point to the same object. Phase 1 returns true immediately.

What to practise. Draw `p1`, `this`, and `obj` as separate reference variables before evaluating the condition.

Phase 1: different names, same object

Setup. Jackie asks about `p1.equals(p4)` under the assumption that `p1` and `p4` are aliases.

Main point. Different variable names do not imply different objects. After the parameter binding, Phase 1 still sees the same stored reference and returns true.

What to practise. Decide the result from the reference diagram, not from the variable names.

Phase 2: null argument

Setup. Page 6 traces `p1.equals(null)`.

Trace skeleton.

1. Dynamic type `PointV2` selects the overridden method.
2. Bind `obj = null`.
3. Phase 1: `this == obj` is false.
4. Phase 2: `obj == null` is true.
5. Return false and bypass the remaining phases.

What to practise. Explain both reasons why an internal `this == null` branch is redundant.

Nine static-type compilation judgments

Setup. Page 8 organizes `o1:C1`, `o2:C2`, and `o3:Object` against `m1`, `m2`, and `equals`.

Main point. Compile-time legality depends on the static type's expectation:

- `o1`: `m1` and `equals` compile; `m2` does not.
- `o2`: `m2` and `equals` compile; `m1` does not.
- `o3`: only `equals` compiles in this example.

What to practise. Rebuild the expectation table from the class hierarchy before checking the calls.

Tracking `o3`: static versus dynamic type

Setup. Page 9 follows `o3` through declaration, `o3 = o1`, and `o3 = o2`.

Main point. The static type remains `Object` in all three states. The dynamic type changes from undefined/null to `C1`, then to `C2`.

What to practise. Maintain separate static-type and dynamic-type columns rather than trying to infer both from one annotation.

Casting and method eligibility

Setup. After `o3 = o1`, direct `o3.m1()` does not compile even though the referenced object is a `C1` object.

Main point. The cast expression `((C1) o3)` has static type `C1`, so `((C1) o3).m1()` compiles. The same cast still cannot call `m2`. After `o3 = o2`, the analogous `C2` cast permits `m2` but not `m1`.

What to practise. For every dot expression, identify the static type of the expression immediately to the left of the dot before deciding whether the method call compiles.

Common Pitfalls and Misconceptions

- **Skipping the `this/obj` setup before tracing `equals`.** Jackie repeatedly asked students to establish the context-object reference and call-by-value parameter binding before line 1.
- **Assuming different reference-variable names mean different objects.** Aliases such as `p1` and `p4` can still satisfy `this == obj`.
- **Checking `this == null` inside an instance method.** A null context reference would have caused `NullPointerException` before the method body, and Phase-2 execution already gives `this != obj`.
- **Using dynamic type to decide whether source code compiles.** Compilation is checked from static type. Even when `o3` dynamically points to a `C1` object, `o3.m1()` does not compile when `o3` is declared as `Object`.
- **Believing reassignment changes static type.** `o3 = o1` or `o3 = o2` changes the referenced object and may change dynamic type, but the declared static type remains `Object`.
- **Treating a cast as a new object or a permanent change to the variable.** Jackie's casting diagram shows the cast as an alias expression to the same object with a different static type for that expression.
- **Assuming a cast grants access to every method.** A cast to `C1` exposes the methods in `C1`'s expectation, not methods such as `m2` that belong to `C2`.
- **Assigning a dynamic type before a reference points to an object.** Jackie clarified that immediately after `Object o3;` in the traced setup, the dynamic type is undefined/null because no object is referenced.
- **Treating the provisional parent/child assignment remark as the full inheritance rule.** Jackie explicitly deferred the full explanation of cross-type assignment compatibility to the inheritance unit and said that deeper rule was not part of Written Test 1.

What You Should Be Able to Do

After this session, you should be able to:

- set up `this` and `obj` correctly before tracing an overridden `equals(Object obj)` call;
- explain and trace Phase 1, `if (this == obj) { return true; }`;
- determine when two differently named reference variables are aliases and therefore return true in Phase 1;
- explain why reaching Phase 2 implies `this != obj`;
- trace Phase 2, `if (obj == null) { return false; }`;
- give both source-supported reasons why checking `this == null` inside the method is redundant;
- distinguish static type from dynamic type and state what each one controls in the current model;
- construct the inherited-method expectation for `Object`, `C1`, and `C2`;
- decide which of the nine `o1`/`o2`/`o3` method-call combinations compile from their static types;
- track `o3`'s static and dynamic types through declaration and reassignment;
- explain why `o3.m1()` can fail to compile even when `o3` dynamically references a `C1` object;
- interpret `((C1) o3)` or `((C2) o3)` using Jackie's cast-as-alias model;
- decide whether a method call on a cast expression compiles from the cast expression's static type; and
- explain what background from this lecture will be used to complete Phase 3 at the beginning of the next class.

Suggested Practice

- Prepare for **Written Test 1 on Wednesday, November 4** using the released guide and practice questions. Test your eCS and Passport York logins beforehand and make sure your Duo setup works on your phone.
- Complete the assigned **Short-Circuit Evaluation** exercise before Written Test 1. Jackie said it takes roughly 20 minutes and is part of the test coverage, even though its detailed development was not part of this lecture.

- Reconstruct pages 3–6 of Jackie's annotated iPad notes without copying the finished arrows. Establish `this` and bind `obj`. Then trace:
  `p1.equals(p1)`, the alias variant, and `p1.equals(null)`.
- Redraw the page-8 `Object/C1/C2` hierarchy and derive the expectation table before checking the nine method calls.
- Redo the page-9 `o3` trace using separate columns for static type and dynamic type.
- Rewatch the final roughly ten minutes of the lecture recording if the casting model is still unfamiliar. Jackie explicitly recommended reviewing this segment.
- For each cast example, say aloud three things: (1) what object is referenced; (2) what the cast expression's static type is; and (3) whether the method belongs to that type's expectation.
- Before Monday, review the casting example carefully because Jackie said Phase 3 and Phase 4 of the overridden `equals` method rely on this background.

Where to Review

- **Jackie's annotated iPad notes, page 2:** Written Test 1, short-circuit exercise, Lab2 due date, and Lab3 release.
- **Jackie's annotated iPad notes, page 3:** the full overridden `PointV2.equals(Object obj)` skeleton, Phase 1, context-object visualization, and call-by-value setup.
- **Jackie's annotated iPad notes, page 4:** Phase-1 trace for `p1.equals(p1)` and the aliasing extension.
- **Jackie's annotated iPad notes, page 5:** Phase 2, the `obj == null` condition, and the annotated redundant alternative checking `this == null`.
- **Jackie's annotated iPad notes, page 6:** trace of `p1.equals(null)` through Phase 1 and Phase 2.
- **Jackie's annotated iPad notes, page 7:** definitions of static type, dynamic type, and type casting, plus the `C1/C2` method-call and cast examples.
- **Jackie's annotated iPad notes, page 8:** inheritance accumulation and the static-type expectation table for `o1`, `o2`, and `o3`.
- **Jackie's annotated iPad notes, page 9:** the tracking table for `o3`'s static and dynamic types across declaration and reassignment.
- **Jackie's annotated iPad notes, pages 10–17:** assigned short-circuit-evaluation material. These pages are preparation material; the detailed short-circuit development was not taught during Lecture 13 itself.

- **Lecture recording, roughly 0:00–0:03:** Lab2/Lab3 logistics, Written Test 1 authentication requirements, the short-circuit exercise, and the explicit coverage boundary.
- **Recording, roughly 0:03–0:06:** recap of default versus overridden `equals` and dynamic-type-based method selection.
- **Recording, roughly 0:06–0:17:** Phase 1, call-by-value visualization, `p1.equals(p1)`, and the alias example.
- **Recording, roughly 0:17–0:32:** Phase 2, the two reasons the `this == null` check is redundant, and the `p1.equals(null)` trace.
- **Recording, roughly 0:32–0:49:** Phase-3 preview, static-type definition, inheritance accumulation, expectation table, and the nine compile-time method-call judgments.
- **Recording, roughly 0:49–0:58:**
  Dynamic-type tracking and the `o3` declaration/reassignment table.
- **Recording, roughly 0:58–1:12:** direct versus cast method calls, the cast-as-alias model, `C1`/`C2` cast examples, and Jackie's Written Test 1 closing guidance.

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 turns each `equals` call into a reference diagram before executing the method body, and how he separates compile-time static-type reasoning from runtime dynamic-type reasoning. If a branch condition, reference arrow, or cast still feels unclear afterwards, return to the recording and replay that exact transition.

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: set up `this` and `obj` and trace Phase 1, explain the two reasons behind Phase 2's null logic, rebuild the `Object`/`C1`/`C2` expectation table, track `o3`'s static and dynamic types, and reconstruct why the `C1` and `C2` cast expressions permit some calls but not others. The goal is not to memorize Jackie's finished arrows, but to be able to reproduce the reference, branch, type, compilation, and casting reasoning from the code.