EECS2030 (B) Advanced Object-Oriented Programming Fall 2026

Digest of Lecture 18, November 16

Polymorphism & Dynamic Binding and Casts: Motivation & Syntax

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 6

Key Examples and Demonstrations 7

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 10

Closing Reminder 11

Main Ideas

- **Lecture 18 reconnects substitution, polymorphism, and dynamic binding before introducing type casting.** The opening examples use the student hierarchy to separate compile-time assignment legality from runtime behaviour.
- **Compilation questions use static types only.** The Rule of Substitution accepts `s = rs` but rejects `rs = s`. Jackie then rebuilds the rejected direction with a contradiction-style object/reference visualization so that the rule can be derived rather than merely memorized.
- **Runtime behaviour uses dynamic types.** A reference variable keeps the same static type after declaration, while a valid reassignment may change its dynamic type. Dynamic binding then selects the overridden method version associated with the current dynamic type.
- **Type casting is motivated by a gap between runtime knowledge and compile-time static typing.** A reference can dynamically point to a more specialized object even when its static type does not expose the subclass-specific members that the object has.
- **A cast creates another alias to the same object.** In Jackie's page-8 visualization, the cast expression has the cast type as its static type. The original variable and object are not changed; a named variable may store the cast result, or the anonymous cast expression may be used directly.
- **Parentheses matter when a casted receiver is followed by a member call.** The final page contrasts a correctly grouped cast-and-call with an ungrouped form in which dot/member access is considered first.
- **The general cast-validity and runtime-exception rules are deliberately deferred.** Lecture 18 establishes the motivation, syntax, alias visualization, named versus anonymous use, and precedence issue. Jackie leaves the general compilation conditions for casts and the runtime `ClassCastException` analysis for the next class.

Roadmap

1. Immediate assessment and study priorities

Jackie's annotated iPad notes page 2 and the opening announcement establish the current priorities:

- **Written Test 2: Wednesday, November 25.** Jackie says material from the upcoming Lecture 19 is included and plans to release a coverage guide plus practice questions on eClass.
- **Lab4: due Wednesday, November 25.** Jackie identifies Lab4 as the current programming priority and says its solution will be provided afterwards.

- **Programming Test 3: the week containing December 2.** The transcript explicitly says the test is in that week and covers Lab4; December 2 is also identified as the last lab session of the semester.

Jackie also asks students to finish and study the asynchronous Lecture 17 before continuing with the inheritance/type-casting material. The Written Test 2 guide release is planned, not guaranteed at a specific time.

2. Review the Rule of Substitution using the student hierarchy

Page 3 returns to the familiar hierarchy with a `Student` reference `s` and a `ResidentStudent` reference `rs`. The first three lines construct separate student and resident-student objects and make the static and dynamic types visible in the object/reference diagram.

Two reversed assignments are then compared:

```
s = rs;
rs = s;
```

The first compiles; the second does not. Jackie repeatedly emphasizes that this is a compilation question, so the judgment is based on the static types on the two sides. The right-hand-side static type must satisfy the left-hand-side expectation through the descendant relationship developed in Lecture 17.

3. Derive why `rs = s` must fail by contradiction

Page 3 gives a second route to the same answer. Jackie temporarily assumes `rs = s` compiles and executes it in the reference diagram.

After that hypothetical assignment, `rs` would point to the same base `Student` object as `s`. However, `rs` still has static type `ResidentStudent`, so its expectation includes the resident-specific `premiumRate`. A line such as `rs.premiumRate` would therefore be accepted based on the static type even though the actual base `Student` object in the diagram does not contain that resident-specific member.

Jackie calls the resulting hypothetical runtime failure a contradiction and uses it to justify why the original assignment is rejected. Treat this as the session's intuition for the rule, not as a new general runtime rule.

4. Polymorphism trace: one static type, changing dynamic types

Page 4 uses a `Student` reference `s`, a resident reference, and a non-resident reference. Jackie marks four checkpoints in a table.

The static type of `s` stays `Student` at every checkpoint. The dynamic type changes with the reference target:

- before an object is assigned to `s`: recorded as `null`;
- after `s = rs`: `ResidentStudent`;
- after the first tuition call and before the next assignment: still `ResidentStudent`;

- after `s = nrs`: the non-resident subclass.

This changing runtime shape of the same statically declared reference is the polymorphism idea Jackie is reviewing.

5. Dynamic binding selects the tuition method version

The same page contains three versions of `getTuition`: the common student version and overridden resident/non-resident versions.

When `s` dynamically points to a `ResidentStudent`, dynamic binding selects the resident version. Jackie illustrates this with a base amount of 100 and a premium rate of 1.25, giving 125. After `s` is reassigned to the non-resident object, the later call uses the non-resident version, illustrated with a 0.75 rate and result 75.

Jackie explicitly says the numerical values are not the important part. The important task is to identify which method version executes from the current dynamic type.

6. Golden rule: static type for compilation, dynamic type for runtime

Page 5 summarizes the review in two lines:

- **Compilation: static types.**
- **Runtime behaviour: dynamic types.**

Runtime examples include which overridden method version executes and, as a preview, whether a cast that compiled can later produce a `ClassCastException`. The exception rule itself is not taught here.

7. Type-cast motivation: the compiler sees only `jim`'s static type

Page 6 begins the type-casting unit. Jackie draws `jim` with static type `Student` but dynamic type `ResidentStudent`. The actual object therefore has resident-specific state and behaviour.

Nevertheless, a resident-specific operation such as `jim.setPremiumRate(1.5)` does not compile because the receiver `jim` is statically a `Student`. Likewise, trying to assign `jim` directly to a `ResidentStudent`-typed reference fails the same static-type compatibility check, even though a human tracing this execution can see that `jim` currently points to a resident-student object.

This mismatch between what the programmer knows at this execution point and what the compiler can use from static types is the motivation for type casting.

8. Why the compiler does not generally infer the final dynamic type

Page 7 presents Jackie's **A+ Challenge**: given an arbitrary Java program and a variable name, produce the variable's last dynamic type.

Jackie uses the challenge to motivate the general limitation. An arbitrary program can contain assignments, conditionals, loops, and recursion. A simple line-by-line simulation may itself fail to terminate if the input program enters an infinite loop or infinite recursion.

Jackie describes the general problem as undecidable and explicitly says the formal proof is outside this course.

The instructional conclusion for Lecture 18 is that the compiler does not decide ordinary compilation by trying to reconstruct arbitrary future runtime dynamic types. It relies on static typing information.

9. Anatomy of a cast expression

Page 8 introduces the cast expression:

```
(ResidentStudent) jim
```

The type in parentheses is the **cast type**. Jackie visualizes the result as an anonymous alias to the same object. Its static type is `ResidentStudent`.

The cast does not alter the original variable's declared static type and does not create or modify the runtime object. The difference lies in the static type of the newly created expression/reference view.

10. Named versus anonymous cast use

The page-8 diagram separates the right-hand-side cast expression from the full assignment. The anonymous cast expression may be used directly, or its value may be stored in a named `ResidentStudent` reference. Jackie draws the original reference, anonymous cast alias, and named resident reference all pointing to the same object.

That distinction explains the member-access results:

- through `jim`: resident-specific access fails because the static type is `Student`;
- through the cast expression: access can compile because the expression is statically `ResidentStudent`;
- through the named resident reference: the same specialized expectation is available.

For a compilation question, identify the static type of the specific expression to the left of the dot.

11. Smartphone example: a cast can expose the needed expectation

Page 9 repeats the idea in the `SmartPhone` hierarchy. A reference `aPhone` has static type `SmartPhone` and dynamically points to an `IPhone13Pro` object.

Calling `aPhone.facetime()` directly does not compile because the static type `SmartPhone` does not contain the `facetime` expectation. After casting the reference to a more specialized type, the resulting expression has a static type that can expose the desired member. The named example on the page stores the cast result in an `IOS`-typed reference; Jackie justifies that assignment using the static types and descendant relationship.

The same runtime object may therefore be reachable through multiple references with different static expectations.

12. Parentheses and operator precedence

The last page compares the correct grouped form with an ungrouped form. Jackie points out that dot/member access has higher precedence than the cast when the receiver is not grouped.

The intended order is:

```
((IPhone13Pro) aPhone).facetime();
```

With the outer parentheses, the cast is applied to `aPhone` first and the member access is checked against the cast expression.

In the ungrouped form, the expression effectively attempts the member access on `aPhone` first. Because `aPhone`'s static type is `SmartPhone`, that member access already fails compilation.

13. Where Lecture 18 stops

Lecture 18 ends after establishing type-cast motivation, syntax, alias visualization, named versus anonymous use, and the parentheses/precedence issue. Jackie explicitly leaves two questions for the next class:

1. Under what conditions does the cast expression itself compile?
2. If it compiles, under what conditions can it still fail at runtime?

`ClassCastException` is named only as a preview, and `instanceof` is mentioned as a possible next-class topic if time permits. Do not infer the full cast rules from Lecture 18 alone.

Key Terms, Notation, and Structures

Static type
The declared type of a reference variable, or the type assigned to an expression by the compiler. It governs the compile-time checks developed in this session and does not change when the variable is reassigned.

Dynamic type
The runtime type of the object currently referenced. It can change after a valid reassignment.

Polymorphism
One fixed static type allowing multiple possible dynamic types during execution.

Dynamic binding
Runtime selection of the applicable overridden method version according to the reference's current dynamic type.

Expectation
The members available through a reference expression according to its static type in Jackie's current inheritance model.

Type cast
An expression such as `(ResidentStudent) jim` that gives the cast expression the static type named in the parentheses.

Cast type
The type written inside the cast parentheses.

Cast expression / alias
Jackie visualizes the cast result as another alias to the same object, with static type matching the cast type.

Named versus anonymous cast use
Store the cast result in a named reference variable, or use the cast expression directly without naming it first.

Operator precedence
The rule determining which operation is considered first. In the final example, dot/member access is considered before an ungrouped cast.

Key Examples and Demonstrations

`s = rs` versus `rs = s`

Setup. Page 3 compares the two assignment directions in the `Student/ResidentStudent` hierarchy.

Main point. `s = rs` compiles, while `rs = s` does not. Jackie then assumes the rejected line compiles and uses the resulting resident-specific expectation on a base `Student` object to reach a contradiction.

What to practise. Give both explanations: the direct static-type rule and the page-3 object/reference contradiction.

Dynamic binding in the tuition trace

Setup. Page 4 tracks `s` through resident and non-resident assignments.

Main point. Static type remains `Student`; dynamic type follows the current object. The selected `getTuition()` version changes with that dynamic type.

What to practise. Recreate the four-checkpoint table and decide separately the static type, dynamic type, and method version.

Three aliases to one resident-student object

Setup. Page 8 begins with `jim` statically typed `Student` but dynamically pointing to a `ResidentStudent`.

Main point. The original reference, anonymous cast expression, and named resident reference can all point to the same object while carrying different static types.

What to practise. Draw the three arrows and label the static type of each expression/reference before deciding member access.

`SmartPhone` cast and `facetime()` access

Setup. Page 9 uses `aPhone` with static type `SmartPhone` and dynamic type `IPhone13Pro`.

Main point. Direct `aPhone.facetime()` fails because the receiver's static type lacks that member. A suitably specialized cast expression or named alias exposes a static expectation that contains it.

What to practise. Underline the receiver to the left of the dot and identify that receiver expression's static type.

Grouped versus ungrouped cast-and-call

Setup. Page 9 compares a parenthesized casted receiver with an ungrouped form.

Main point. The outer parentheses force the cast to occur before the member access. Without them, the member access is checked on the original `SmartPhone`-typed receiver first.

What to practise. Mark the operator evaluation order character by character before judging compilation.

Common Pitfalls and Misconceptions

- **Using dynamic type to answer a compilation question.** Jackie repeatedly says that the compilation checks in this lecture use static types only.
- **Reversing the Rule of Substitution.** The student-to-resident direction is not automatically legal just because the resident-to-student direction is legal.
- **Memorizing the assignment direction without understanding why.** Jackie explicitly provides the page-3 contradiction so students can rebuild the reasoning from expectation and object structure.
- **Assuming a method call changes dynamic type.** In the page-4 trace, the first tuition call leaves `s`'s dynamic type unchanged; the later reassignment changes it.
- **Treating a subclass object as exposing subclass-only members through a superclass-typed reference.** The page-6 and page-9 examples show that member access is still checked against the receiver expression's static type.

- **Thinking a cast changes the original variable or object.** The cast is visualized as another alias; the original variable's declared static type and the runtime object remain unchanged.
- **Assuming every cast is automatically legal and safe.** Jackie explicitly defers both cast compilation rules and runtime `ClassCastException` conditions to the next class.
- **Forgetting the outer parentheses around a casted receiver.** The final example shows that dot/member access is considered first in the ungrouped form.
- **Turning the A+ challenge into a formal theorem or complete proof.** Jackie uses it as an informal undecidability motivation and explicitly says the formal proof is outside the course.

What You Should Be Able to Do

After this session, you should be able to:

- state the November 25 Written Test 2 and Lab4 dates and identify Lab4 as the stated coverage for Programming Test 3;
- explain why Lecture 17 should be reviewed before continuing the type casting unit;
- apply the Rule of Substitution to the two assignment directions using static types;
- reconstruct Jackie's contradiction-style explanation for why `rs = s` is rejected;
- trace a fixed static type and changing dynamic type through the page-4 assignment sequence;
- determine which overridden `getTuition()` version runs from the current dynamic type;
- distinguish compilation reasoning from runtime reasoning using the page-5 static-versus-dynamic-type rule;
- explain why general runtime dynamic-type inference is not something the compiler relies on for ordinary compilation checks;
- explain the motivation for a type cast in the `jim` example;
- identify the cast type and static type of a cast expression;
- draw the original reference, anonymous cast alias, and named cast reference pointing to the same object;
- decide subclass-specific member access from the receiver expression's static type;
- distinguish named and anonymous use of a cast result; and
- explain why the grouped and ungrouped cast/member-access expressions on page 9 have different compilation behaviour.

Suggested Practice

- **Prioritize Lab4**, due Wednesday, November 25. Jackie also says Programming Test 3 will cover Lab4.
- Finish and review asynchronous Lecture 17 before the next class, especially expectation, the Rule of Substitution, polymorphism, and dynamic binding.
- Redraw the page-3 `rs = s` contradiction from scratch. Do not stop at "wrong direction"; explain the static expectation and the object mismatch Jackie uses to derive the contradiction.
- Reproduce the page-4 four-checkpoint table for `s`. Keep static type, dynamic type, and selected `getTuition()` version separate.
- Review the page-5 golden rule until you can immediately classify a new question as compilation-time or runtime reasoning.
- Reconstruct page 8 with the original student reference, the anonymous resident cast expression, and the named resident reference. Label the static type of each alias.
- On page 9, compare the direct, named-cast, and anonymous-cast uses of `facetime()` by identifying the receiver expression and its static type first.
- Check the outer parentheses in the final cast/member-access exercise character by character before deciding which operation is considered first.
- Do not try to supply the complete cast-legality or `ClassCastException` rules from this lecture. Jackie explicitly leaves them for the next class.

Where to Review

- **Jackie's annotated iPad notes, pages 1–2:** Lecture 18 topic map, Written Test 2/Lab4 reminders, Programming Test 3 reminder, and the asynchronous Lecture 17 review priority.
- **Jackie's annotated iPad notes, page 3:** substitution recap, `s = rs` versus `rs = s`, static expectations, and the contradiction-style object/reference visualization.
- **Jackie's annotated iPad notes, page 4:** polymorphism/dynamic binding trace table, resident/non-resident objects, and `getTuition()` dispatch.
- **Jackie's annotated iPad notes, page 5:** compilation uses static types; runtime behaviour uses dynamic types; preview of `ClassCastException`.
- **Jackie's annotated iPad notes, pages 6–7:** type-cast motivation and the A+ undecidability challenge used to explain compiler limits.
- **Jackie's annotated iPad notes, page 8:** anatomy of a type cast, cast type, anonymous alias, named reference, and resident-specific member-access comparison.

- **Jackie's annotated iPad notes, page 9:** the `SmartPhone/IPhone13Pro` example, named versus anonymous casts, `IOS`-typed alias, and operator precedence.
- **Lecture recording, roughly 0:00–0:03:** assessment calendar, Lab4 priority, Programming Test 2 grading update, and Lecture 17 review reminder.
- **Recording, roughly 0:03–0:19:** substitution recap, `rs = s`, static expectation, and the contradiction-style argument.
- **Recording, roughly 0:19–0:33:** polymorphism trace, `getTuition()` dynamic binding, and the static-type versus dynamic-type golden rule.
- **Recording, roughly 0:33–0:50:** type-cast motivation, compiler limitations, the A+ challenge, and the undecidability/nontermination intuition.
- **Recording, roughly 0:50–1:07:** cast anatomy and alias visualization, the smartphone casting example, named versus anonymous use, operator precedence, and the next-class preview.

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 separates compile-time reasoning from runtime reasoning, updates object/reference diagrams, and moves from substitution and dynamic binding into the motivation for type casting. If a transition, explanation, cast expression, reference arrow, or method-selection step remains unclear afterwards, 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, choose a suitable selection of the important reasoning from this session and work through it again on your own: reconstruct the invalid `rs = s` contradiction, redo the static/dynamic-type table and `getTuition()` dispatch, draw the three-reference cast-alias picture, and compare the grouped and ungrouped cast/member-access expressions character by character. The goal is not to memorize Jackie's recorded steps, but to determine whether you can reproduce the compile-time and runtime reasoning from your own understanding and explain why the important steps work.