EECS2030 (B) Advanced Object-Oriented Programming Fall 2026

Digest of Lecture 19, November 18

Type Casting: Compilation, Runtime Safety, and ClassCastException

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 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 10

Closing Reminder 11

Main Ideas

- **Type-cast reasoning has two separate stages.** First ask whether the cast **compiles**; only after that ask whether the cast **executes without an exception at runtime**. Jackie repeatedly separates these questions because they use different kinds of type information.
- **Compilation uses static type and the inheritance hierarchy.** In the hierarchy model developed in this session, a cast from a reference's static type to one of its ancestors is an **upward cast**; a cast to one of its descendants is a **downward cast**. Those casts compile. A target that is neither ancestor nor descendant does not.
- **Upward and downward casts change expectation, not the object.** An upward cast narrows the static expectation; a downward cast widens it. The cast expression is still an alias to the same runtime object.
- **A compilable cast can still fail at runtime.** Runtime analysis uses the object's current dynamic type. The cast is exception-free only when the dynamic type can fulfill the expectation of the cast type; otherwise Java throws `ClassCastException` at the cast.
- **Page 9 combines the two stages into one classification.** With static type `Android` and dynamic type `Samsung`, the hierarchy is first divided into compilable versus non-compilable cast targets, then the compilable targets are divided into exception-free casts versus casts that throw `ClassCastException`.
- **Chained casts can change what compiles without changing the runtime object.** Page 10 shows that a direct cast from static type `B` to unrelated type `D` does not compile, while the sequence `B -> A -> D` can compile through the common ancestor. Runtime must still be checked cast by cast; the second cast fails for the shown object of dynamic type `C`.

Roadmap

1. Assessment priorities and the Written Test 2 endpoint

Jackie's annotated iPad notes page 2 lists:

- **Written Test 2: Wednesday, November 25;**
- **Lab4: due Wednesday, November 25;**
- **Programming Test 3: Wednesday, December 2.**

Jackie also said the Written Test 2 guide and practice questions had been released, and pointed students to a required-reading item under the inheritance section covering static type, expectation, dynamic type, and type casting.

The transcript gives a precise Written Test 2 coverage endpoint: **through slide 67**. The page-10 exercise in Jackie's annotated iPad notes is labelled "Inheritance: Slide 67" and is the final worked item of this session's tested material.

2. The session roadmap: compile first, runtime second

Page 3 reduces type-cast analysis to two questions:

1. When does a type cast compile? **Use static types.**
2. Given that the cast compiles, when does it execute without an exception at runtime? **Use dynamic type.**

Jackie previews `instanceof` as a later third topic. It is not developed in this lecture. Jackie explicitly says `instanceof` is not covered on Written Test 2, although it will matter later for Lab4 and the exam.

3. Compile-time cast rule from the smartphone hierarchy

Page 4 starts with `myPhone` whose static type is `Android`. For the compilation question, Jackie explicitly crosses out the dynamic-type information: it is irrelevant at this stage.

Relative to `Android`:

- casting upward to an ancestor such as `SmartPhone` compiles;
- casting downward to a descendant such as `GalaxyS21Plus` compiles;
- casting to a type on the unrelated `IOS` branch does not compile.

Jackie names the first case an **upward cast** and the second a **downward cast**. Casting to the same type is also possible but redundant.

4. Expectation explains what upward and downward casts do

The expectation table on page 4 compares three aliases to the same conceptual object:

- `myPhone` with static type `Android`;
- `sp` after an upward cast to `SmartPhone`;
- `ga` after a downward cast to `GalaxyS21Plus`.

The upward cast narrows expectation: moving from `Android` to `SmartPhone` removes Android-specific members from what can be called through that alias.

The downward cast widens expectation: moving from `Android` to `GalaxyS21Plus` gives the cast expression the more specialized static expectation shown on the descendant path.

Crucially, Jackie says these are aliases to the **same object**. Casting changes the static type through which the object is viewed; it does not create a new object.

5. A cast can compile and still fail at runtime

Page 5 switches to the student hierarchy:

`Student` ⟶ `ResidentStudent`, `NonResidentStudent`.

The code begins with a `Student`-typed reference `jim` pointing to a `NonResidentStudent` object. The next line casts `jim` to `ResidentStudent` and stores the result in `rs`.

At compile time, this is a downward cast from `Student` to its resident-student descendant, so it compiles. The resulting cast expression has static type `ResidentStudent`.

At runtime, however, the object is dynamically a `NonResidentStudent`. Jackie asks the decisive question: **Can the dynamic type fulfill the expectation of the cast type?**

Here the answer is no, so Java throws `ClassCastException` at the cast.

6. Why the exception occurs at the cast, not on the later member call

Page 5 also shows a later call to `rs.setPremiumRate(1.5)`. Jackie uses that line only to explain what would eventually go wrong if the invalid runtime cast were allowed to continue.

Actual execution does **not** reach that line. The exception is raised earlier, at the cast itself, because the runtime system detects that the `NonResidentStudent` object cannot fulfill the `ResidentStudent` expectation.

Jackie frames this as an early-error principle: detect the incompatible cast at the point where the invalid assumption is introduced rather than allowing a later crash.

7. A second runtime-failure example in the smartphone hierarchy

Page 6 gives:

```
SmartPhone aPhone = new GalaxyS21Plus();
IPhone13Pro forHeeyeon = (IPhone13Pro) aPhone;
forHeeyeon.quickTake();
```

The downcast from `SmartPhone` passes compilation. At runtime, the dynamic type is `GalaxyS21Plus`. The cast type is `IPhone13Pro`. The dynamic object cannot fulfill that cast-type expectation, so Java throws `ClassCastException`. The later `quickTake()` call is not reached.

Pages 7–8 leave closely related smartphone cases for students to analyse with the same two-stage reasoning.

8. General runtime rule: compare dynamic type with cast type

After the examples, Jackie generalizes the runtime question.

For a cast that already compiles:

> Can the current dynamic type fulfill the expectation of the cast type?

In the hierarchy language used throughout the course, the cast is **exception-free** when the cast type is an ancestor of the dynamic type (or the same type). Equivalently, the dynamic type is a descendant of the cast type.

If that relation does not hold, the cast throws `ClassCastException`.

Jackie also clarifies that `ClassCastException` is a built-in Java exception; students do not define it themselves.

9. Page 9: combine compile-time and runtime classifications

Page 9 is the session's summary diagram. It fixes:

$$\mathrm{ST}(\texttt{myPhone}) = \texttt{Android}, \qquad \mathrm{DT}(\texttt{myPhone}) = \texttt{Samsung}.$$

Stage 1: compilation. Using only static type `Android`, the compilable targets are `Android` itself, its ancestors, and its descendants. Types on the `IOS` branch are marked non-compilable.

Stage 2: runtime. Now consider only the cast targets that compiled. Since the dynamic type is `Samsung`, exception-free targets lie on the ancestor path of `Samsung`, including `Samsung` itself. Other compilable downward targets, such as the Huawei branch, are shown as `ClassCastException` cases because their expectations cannot be fulfilled by the dynamic type.

This page is Jackie's recommended starting point for reviewing the big picture for Written Test 2 and the exam.

10. Build and verify the hierarchy before answering cast questions

Page 10 begins the final exercise from four declarations:

```
class A { }
class B extends A { }
class C extends B { }
class D extends A { }
```

Jackie's first instruction is to build the hierarchy and double-check it before answering the rest. The diagram has three `extends` relationships:

$$\texttt{A} \leftarrow \texttt{B} \leftarrow \texttt{C}, \qquad \texttt{D extends A}.$$

Jackie warns that an incorrect hierarchy can compromise many later answers.

11. Direct cast from `B` to `D`: does not compile

The exercise first creates a `B`-typed reference pointing to a `C` object. That initialization compiles because `C` is a descendant of `B`.

A later direct cast tries to cast that `B`-typed expression to `D`. For the compilation question, only the static type matters: `D` is neither an ancestor nor a descendant of `B`. Therefore the cast itself does not compile.

Jackie explicitly asks students to identify **which part** fails – the cast, not merely "the whole line" – and to write a precise justification.

12. Chained cast through a common ancestor

The same page then rewrites the idea as a sequence:

$$\texttt{B} \longrightarrow \texttt{A} \longrightarrow \texttt{D}.$$

The first cast is upward from `B` to ancestor `A`; it compiles. That cast expression now has static type `A`. From there, the cast to descendant `D` is a downward cast and also compiles. The resulting expression has static type `D`, so assignment to a `D`-typed variable also compiles.

Jackie generalizes the **compile-time pattern** by pointing out that `Object` is a common ancestor of Java classes, so an otherwise unrelated pair can be routed upward to `Object` and then downward.

This is only a compilation route. It is **not** a guarantee of runtime success.

13. Runtime trace of the chained cast

Runtime analysis returns to the actual object. The `B`-typed reference points to an object of dynamic type `C`.

For the first cast to `A`:

- dynamic type is `C`;
- cast type is `A`;
- `C` is a descendant of `A`;
- therefore the first cast is exception-free.

The cast expression remains an alias to the same `C` object; only its static type becomes `A`. For the second cast to `D`:

- the dynamic type is still `C`;
- the cast type is `D`;
- `C` is not a descendant of `D`;
- therefore the second cast throws `ClassCastException`.

The chained expression demonstrates the full distinction: **compilation can be made possible through an intermediate type, but every runtime cast must still be safe for the actual object.**

14. Where Lecture 19 stops

Jackie closes by summarizing the cast workflow: syntax/alias visualization, when a cast compiles, and when a compilable cast throws `ClassCastException`.

The next inheritance topic is `instanceof`. Jackie says it is not part of Written Test 2 but will be needed for Lab4 and should be studied for the exam. Jackie also briefly previews later inheritance work involving polymorphic arrays and a possible return to call-by-value, followed by recursion in the final lectures.

Key Terms, Notation, and Structures

Upward cast
A cast from the current static type to one of its ancestors. In this session, Jackie describes it as narrowing/reducing expectation.

Downward cast
A cast from the current static type to one of its descendants. Jackie describes it as widening/extending expectation.

Cast type
The class named inside the cast parentheses. If the cast compiles, the cast expression has this static type.

Cast expression
The result of applying a cast to a reference expression. It remains an alias to the same object; its static type is the cast type.

Compilable cast
In the hierarchy model used here, a cast whose target is the current static type itself, an ancestor, or a descendant. The compilation check uses static type only.

Exception-free cast
A compilable cast for which the current dynamic type can fulfill the cast type's expectation. Jackie expresses this as the cast type being on the dynamic type's ancestor path.

`ClassCastException`
The built-in runtime exception thrown when a compilable cast is incompatible with the actual object's dynamic type.

Static type
The declared type used for compile-time cast analysis. It does not change during execution.

Dynamic type
The runtime class of the object currently referenced. It is used for the runtime safety check after compilation succeeds.

Expectation
The attributes/methods promised by a static type in Jackie's inheritance model. Cast reasoning asks whether the actual dynamic object can fulfill the expectation of the cast type.

Key Examples and Demonstrations

Page 4: one object viewed through different static types

Setup. Start from `myPhone` with static type `Android`; compare an upward cast to `SmartPhone` with a downward cast to `GalaxyS21Plus`.

Main point. The aliases have different static expectations, but casting does not create a new object.

What to practise. Reconstruct the expectation table and explain which methods disappear in the upward view and which additional expectation appears in the downward view.

Page 5: compilable cast that fails before line 3

Setup. `jim` has static type `Student` and dynamic type `NonResidentStudent`; the code casts to `ResidentStudent`.

Main point. The downward cast compiles, but at runtime the dynamic type cannot fulfill the cast type's expectation. `ClassCastException` is thrown at the cast, so the later premium-rate call is not executed.

What to practise. Separate the explanation into compile-time and runtime parts rather than giving one combined answer.

Page 9: two-layer classification of a whole hierarchy

Setup. Static type is `Android`; dynamic type is `Samsung`.

Main point. First classify all hierarchy nodes by compilation using static type. Then, among only the compilable targets, classify runtime safety using the dynamic type's ancestor path.

What to practise. Recreate the two regions independently before comparing your result with Jackie's coloured diagram.

Page 10: direct cast versus chained cast

Setup. `B extends A`, `C extends B`, and `D extends A`; a `B`-typed reference points to a `C` object.

Main point. Direct `B -> D` does not compile. The sequence `B -> A -> D` compiles, but at runtime the second cast fails because the actual object is still dynamically `C`.

What to practise. Label the static type after each cast expression and the unchanged dynamic type of the object, then check each runtime cast one at a time.

Common Pitfalls and Misconceptions

- **Using dynamic type to decide whether a cast compiles.** Jackie repeatedly says the compilation question uses static type only.
- **Assuming a compilable cast must work at runtime.** Pages 5–9 deliberately show downward casts that compile but later throw `ClassCastException`.
- **Tracing past a cast that has already thrown.** In the student example, line 3 is only a hypothetical explanation of the later contradiction. Actual execution stops at the failing cast on line 2.

- **Treating a cast as object creation.** The cast expression is another alias to the same object; the object's dynamic type does not change.
- **Analyzing runtime for a cast that does not compile.** Page 9 removes those targets first. Runtime analysis applies only after the static-type compilation check succeeds.
- **Saying only that "the line does not compile."** Jackie asks students to identify whether the cast itself or the later assignment is the failing part and to justify it precisely.
- **Using an unchecked inheritance diagram.** Jackie warns that a wrong hierarchy can make many later answers wrong; build and double-check it first.
- **Interpreting the `Object` bridge as a runtime guarantee.** Routing through a common ancestor can provide a compile-time cast sequence. The actual object still has to pass every runtime cast check.
- **Trying to use `instanceof` as if it were already part of this lecture's tested material.** Jackie previews it for the next inheritance lecture and explicitly excludes it from Written Test 2.

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 the December 2 Programming Test 3 date;
- identify slide 67 as the spoken Written Test 2 coverage endpoint;
- separate cast compilation from runtime cast safety before beginning an analysis;
- determine whether a cast is upward, downward, or neither using the reference expression's static type and the hierarchy;
- explain how an upward cast narrows expectation and a downward cast widens expectation;
- explain why a cast expression is an alias rather than a new object;
- identify the cast expression's static type from the cast type;
- determine whether a compilable cast is exception-free by comparing the current dynamic type with the cast type;
- explain when and why `ClassCastException` is thrown;
- stop a runtime trace at the failing cast rather than continuing to later statements;
- classify an entire hierarchy into compilable/non-compilable targets and then exception-free/`ClassCastException` targets;
- construct and verify the `A`/`B`/`C`/`D` hierarchy from `extends` declarations;

- distinguish a direct non-compilable cast from a chained cast through a common ancestor; and
- trace each cast in a chain separately at runtime while preserving the object's dynamic type.

Suggested Practice

- Read the required item Jackie pointed to under the lecture site's inheritance section and use the released Written Test 2 guide/practice questions.
- Recreate page 4's smartphone expectation table from the hierarchy rather than copying the finished colouring. Explain the upward/downward expectation changes in words.
- Try the page-5 `ClassCastException` code in Java as Jackie recommends. Confirm that execution stops at the cast rather than reaching the later premium-rate call.
- Retry the page-7/page-8 smartphone exercises independently. For every cast, answer "compiles?" before "runtime exception?"
- Use page 9 as a blank classification exercise: mark compilable targets from the static type first, then overlay exception-free targets from the dynamic type.
- Rebuild the page-10 hierarchy from its four class declarations and double-check all three `extends` relationships before analysing any cast.
- Practise writing Jackie's concise justification style: identify the source static type, cast type, and ancestor/descendant relationship rather than writing only "compiles" or "does not compile."
- Reproduce the chained-cast runtime trace one cast at a time. Keep the object's dynamic type `C` fixed while the cast expressions' static types change.
- Do not study `instanceof` as part of Written Test 2 coverage. Jackie says it will be introduced next and should be studied later for the exam and Lab4.

Where to Review

- **Jackie's annotated iPad notes, pages 1–3:** Lecture 19 topic map, November 25/December 2 reminders, and the two-stage type-cast roadmap.
- **Jackie's annotated iPad notes, page 4:** upward versus downward casts, the smartphone hierarchy, aliases, and the expectation table.
- **Jackie's annotated iPad notes, pages 5–6:** compilable casts that fail at runtime in the student and smartphone examples.
- **Jackie's annotated iPad notes, pages 7–8:** independent smartphone exercises asking both "Compilable?" and "`ClassCastException` at runtime?"

- **Jackie's annotated iPad notes, page 9:** the integrated compilable-cast versus exception-free-cast classification with static type `Android` and dynamic type `Samsung`.
- **Jackie's annotated iPad notes, page 10:** slide-67 `A/B/C/D` hierarchy, direct cast, chained cast, static-type changes, and runtime failure.
- **Lecture recording, roughly 0:00–0:06:** Written Test 2 logistics, required reading, slide-67 coverage endpoint, and the two cast questions.
- **Recording, roughly 0:06–0:21:** smartphone hierarchy, compile-time upward/downward rules, aliases, and expectation changes.
- **Recording, roughly 0:21–0:38:** student and smartphone runtime failures and the two-stage `ClassCastException` recap.
- **Recording, roughly 0:38–0:47:** page-9 two-stage hierarchy classification for compilation and runtime safety.
- **Recording, roughly 0:47–1:08:** slide-67 hierarchy construction, the direct and chained cast cases, the `Object` bridge, and the step-by-step runtime trace.
- **Recording, roughly 1:08–1:11:** final type-cast recap and preview of `instanceof`, polymorphic arrays, and later inheritance topics.

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 cast decisions from runtime cast decisions, builds the inheritance hierarchy before reasoning from it, and keeps static type, dynamic type, cast type, and object identity distinct. If a transition, classification, hierarchy relation, or runtime trace 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: rebuild the smartphone cast classification, explain one `ClassCastException` trace from dynamic type and cast type, construct and verify the `A/B/C/ D` hierarchy, and trace the chained cast through its compile-time and runtime stages. The goal is not to memorize Jackie's recorded arrows or colours, but to determine whether you can reproduce the cast reasoning from the hierarchy, static types, dynamic type, and expectations.