EECS2030 (B) Advanced Object-Oriented Programming Fall 2026

Digest of Lecture 16, November 9

Inheritance Design, super, and Static-Type Expectations

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 8

Common Pitfalls and Misconceptions 9

What You Should Be Able to Do 10

Suggested Practice 11

Where to Review 11

Closing Reminder 12

Main Ideas

- **Inheritance is introduced as a design response to duplication and fragmented collections.** In Design Attempt 2, common behaviour such as `register` appears in multiple unrelated classes, so one policy change requires multiple edits. The same separation also leads to multiple student arrays and multiple processing loops.
- **Design Attempt 3 factors shared members into a common parent.** `ResidentStudent` and `NonResidentStudent` both `extend Student`. Common non-private state and behaviour are declared once in `Student`; child-specific state remains in the relevant subclass.
- **A child may inherit a method unchanged or override it.** The common `register` method is inherited without redeclaration. Both child classes override `getTuition`, while `super.getTuition()` reuses the parent's base-tuition calculation rather than duplicating its loop.
- `super` **refers to the immediate parent in the examples developed here.** Jackie uses `super.getTuition()` for a parent-method call and `super(...)` for a parent-constructor call. She also emphasizes that the constructor call must be the first statement in the child constructor and that `super` cannot be chained to skip directly to a grandparent.
- **Runtime object structure and compile-time expectation answer different questions.** Runtime object boxes are labelled by dynamic type and contain the inherited plus child-specific attributes. By contrast, the **expectation** of a reference variable – the range of attributes/methods callable without a compilation error – is determined by its static type.
- **For multilevel inheritance, follow the ancestor path upward.** The smartphone hierarchy shows how a static type accumulates inherited expectations from its class and ancestors. Higher common ancestors explain broadly shared methods; closer common ancestors explain branch-specific sharing; leaf classes can introduce their own unique methods.

Roadmap

1. Immediate priorities: Programming Test 2 and Wednesday's asynchronous lecture

Jackie's annotated iPad notes list **Programming Test 2 on Wednesday, November 11**. The guide and practice questions, Lab3 solution, and a review session for PracticeTest2 have been released.

The ordinary lecture meeting on November 11 will be **asynchronous rather than in person**. Programming Test 2 still takes place that Wednesday during the in-lab session. Jackie said the asynchronous lecture materials would be posted later on November 9 or early on November 10 and asked students not to procrastinate because the upcoming inheritance material is important.

The transcript also states that **Lab3 is covered by Programming Test 2** and that its official submission deadline is the same day as the test, November 11. No submission time was stated in this lecture.

For continued Written Test 2 and final-exam preparation, Jackie also reminded students about the recent assigned exercises, especially reference versus object equality, tracing `equals/assertEquals`, and SMS Design Attempt 1.

2. Finish Design Attempt 2: use maintenance to test the Single Choice Principle

Pages 3–4 return to the no-inheritance design from Lecture 15. The resident and non-resident student classes contain duplicated logic.

Jackie's test for the Single Choice Principle is concrete: propose a policy change and count how many places must be modified. The registration example adds a new maximum-course constraint. Because both classes have their own `register` method, the same policy must be changed in two places rather than the minimum possible one.

Jackie says the same style of reasoning can be applied to duplicated tuition logic. For a test or exam, the exact classes may change; the important skill is to explain the design principle from the maintenance consequences rather than memorize this particular code.

3. Separate arrays and loops reveal another limitation

Page 5 introduces the student-management-system class for Design Attempt 2. Because `ResidentStudent` and `NonResidentStudent` are unrelated classes in that design, the system keeps separate arrays, counters, and loops.

Jackie uses `registerAll` to show the scaling problem. Two student kinds mean two arrays and two loops; five kinds would mean five arrays and five loops under the design being discussed.

The target idea is a **single polymorphic collection of students**, regardless of how many student kinds exist. Jackie explicitly says inheritance will make that design possible later, but the substitution rules that justify the collection are not developed in this lecture.

4. Design Attempt 3: establish the inheritance hierarchy with `extends`

Page 6 introduces:

```
class ResidentStudent extends Student
class NonResidentStudent extends Student
```

Jackie's terminology is:

- the class before `extends` is the child/subclass;
- the class after `extends` is the parent/superclass.

The parent `Student` class holds the common student state and behaviour: name, registered courses, number of courses, `register`, and a base `getTuition` calculation. The child classes add specialized state. `ResidentStudent` adds `premiumRate`; `NonResidentStudent` adds `discountRate`.

5. Factor common members upward; keep specialized members downward

Jackie repeatedly connects the placement of members to the two design principles from the previous lecture.

Single Choice Principle. Common non-private members belong in the parent so that shared behaviour is declared once and inherited rather than duplicated in every child.

Cohesion. Members specific to one child belong in that child. A resident-only premium rate does not belong in every student object; a non-resident-only discount rate does not belong in every student object.

Jackie explicitly cautions students to reason about the design principles rather than assume that the mere presence of inheritance automatically makes a design good.

6. Inheritance in this session means non-private parent members

Jackie summarizes the current rule as: the parent's **non-private** attributes and methods are inherited by the child classes. She later previews public, package-level, and protected access for the asynchronous Wednesday material.

The common `register` method illustrates inheritance without overriding: the child classes can use the inherited implementation without redeclaring it.

The lecture does not develop a complete visibility model here. Keep the current rule within the scope in which Jackie presents it and return to the Wednesday material for the modifier details.

7. Override `getTuition` while reusing the common calculation

The parent `Student.getTuition` calculates the base tuition by summing the fees of the registered courses. Both child classes need a specialized result, so each redeclares the method and therefore overrides the inherited version.

Rather than copying the parent's fee-summing loop, the child implementation can reuse it:

```
double base = super.getTuition();
```

The resident version then applies the premium rate; the non-resident version applies the discount rate. Jackie explicitly presents this as a two-phase calculation: reuse the common base calculation, then perform the child-specific adjustment.

8. Distinguish `this`, `super.m()`, and `super(...)`

Jackie contrasts the current-object and parent-class roles:

- `this` refers to the current object/context;
- `super.getTuition()` invokes the immediate parent's method version;
- `super(...)` invokes the immediate parent constructor.

For the constructor form, Jackie gives an explicit restriction: `super(...)` must be the **first statement** in the child constructor.

A student also asked whether a Java class can have more than one superclass. Jackie's answer for class inheritance is no: a class can `extend` only one class. She contrasts this with interfaces, which are deferred to later course material.

9. Multilevel inheritance: `super` moves up one level

Page 7 uses a three-level hierarchy:

$$\texttt{P} \leftarrow \texttt{C} \leftarrow \texttt{GC}.$$

Method `m` is overridden in successive levels. Inside `GC`, `super.m()` refers to the immediate parent `C`'s version, not directly to `P`'s version.

Jackie explicitly asks students to cross out the idea of chaining `super` to skip a level. She describes that attempt as a compilation error. The practical tracing rule for this lecture is simple: resolve `super` through the direct parent relationship.

10. Four design tips: one parent rule plus 2A/2B/2C for a child

Page 8 summarizes the design guidance.

1. Parent level. Declare attributes and methods common to all subclasses in the parent to minimize duplication and support the Single Choice Principle.

2A. Child level: inheritance. The child receives the parent's non-private members.

2B. Child level: specialization. Add attributes or methods unique to that child; this supports cohesion.

2C. Child level: overriding. Redefine an inherited method when specialized behaviour is needed, and reuse the parent version with `super` when appropriate.

Jackie calls these four points the practical tips to keep in mind when programming with inheritance.

11. Visualize parent and child objects from the hierarchy

Pages 9–10 return to runtime object/reference diagrams. The object box is labelled by its **dynamic type**. The fields in the box come from the accumulated attributes on that object's inheritance path.

Accordingly:

- a `Student` object contains the common student attributes;
- a `ResidentStudent` object contains those common attributes plus `premiumRate`;
- a `NonResidentStudent` object contains those common attributes plus `discountRate`.

Jackie then compares the runtime pictures from Design Attempt 2 and Design Attempt 3. They can look **identical**. Runtime object structure alone does not prove that inheritance is being used; inspect the class declarations for `extends`.

12. Expectation is a compile-time question determined by static type

Page 11 revisits a term introduced earlier in the equality unit: **expectation**. Jackie defines the expectation of a reference variable `x` as the range of attributes and/or methods callable on `x`.

The central rule is:

$$\boxed{\text{the static type of } x \text{ determines the expectation of } x}$$

Jackie explicitly rejects dynamic type for this compile-time question.

The table has one row for each static type. The parent row uses `Student`. One child row uses `ResidentStudent`; the other uses `NonResidentStudent`. Parent-level members are in all three expectations. Each child adds only its own specialized members; sibling-specific members do not cross over.

13. Practise expectation from static type before moving to substitution

Jackie calls the expectation table important and points students to a fresh lecture-site exercise containing a hierarchy and blank table. Her instruction is to practise the expectation classification at least once.

This is preparation for the next step of the inheritance unit. Jackie says the asynchronous Wednesday material will develop deeper from expectation toward substitution. Those substitution rules are not supplied or inferred in this digest.

14. Multilevel expectation: follow the ancestor path in the smartphone hierarchy

Page 12 uses a multilevel hierarchy whose top class is `SmartPhone`. It branches through `IOS` and `Android`, then through brands/models.

To determine the expectation of a static type, Jackie recommends following its **ancestor path** upward and accumulating the methods encountered along the way.

The visual illustrates three levels of sharing:

- `dial` and `surfWeb` come from the top `SmartPhone` level and are common expectations;
- `skype` is shared by the Android branch through the closest common ancestor of the Huawei and Samsung examples;
- leaf/brand-specific methods such as `quickTake`, `zoomage`, and `sideSync` appear only on their corresponding paths.

Jackie also notes that `surfWeb` is overridden at the IOS and Android levels; it remains the same method in the expectation even though the runtime implementation can differ.

15. Where Lecture 16 stops

The lecture stops after the smartphone expectation comparison. Jackie closes by reminding students that Programming Test 2 still runs on Wednesday even though the ordinary lecture is asynchronous, and that the asynchronous inheritance material should be studied before the class reconvenes.

The next material will deepen modifiers, expectation, and substitution. This digest therefore stops at the source-grounded static-type/expectation reasoning developed in Lecture 16.

Key Terms, Notation, and Structures

Inheritance
The parent/child class structure used here to factor common state and behaviour into one parent while allowing specialized child classes.

`extends`
Java keyword that establishes the class relationship. In `Child extends Parent`, the class before `extends` is the child/subclass and the class after it is the parent/superclass.

Inherited member
In the model used in this lecture, a non-private parent attribute or method that becomes available to the child class.

Overriding
Redeclaring an inherited method in a child to provide a child-specific version. The session uses `getTuition` as the main example.

`super.m()`
Calls method `m` from the immediate parent class.

`super(...)`
Invokes the immediate parent constructor. Jackie emphasizes that this call must be the first statement in the child constructor.

Static type
The declared type of a reference variable. It determines compile-time expectation.

Dynamic type
The runtime class of the object. Jackie uses it to label object boxes and to reason about runtime selection among overridden method versions.

Expectation
Jackie's term for the range of attributes and/or methods callable through a reference variable. The static type determines the expectation.

Ancestor path
The upward path from a class through its parent, grandparent, and further ancestors. Jackie uses this path to accumulate expectation in a multilevel hierarchy.

Closest common ancestor
The nearest shared ancestor on two class paths. In the smartphone example, it explains why two branches can share methods that another branch does not.

Key Examples and Demonstrations

Duplicated `register`: count maintenance locations

Setup. Page 3 shows two unrelated student classes, each with its own `register` method.

Main point. A new registration rule must be added to both copies, so the change occurs in two places rather than the minimum possible one. Jackie uses this as the concrete Single Choice Principle violation.

What to practise. Do not stop at "there is duplication." State a specific requirement change, identify every place affected, and explain why the count is not minimal.

A single polymorphic collection as design motivation

Setup. Page 5 shows separate resident/non-resident arrays, counters, and loops in the no-inheritance management system.

Main point. More unrelated student kinds mean more separate arrays and processing loops. The inheritance design motivates one polymorphic collection, but the substitution rules behind that collection are intentionally left for later.

What to practise. Compare the two-array/two-loop structure with the target one-collection idea without inventing the later substitution rules.

Two-phase tuition calculation with parent reuse

Setup. Page 6 gives a parent `getTuition` plus overridden resident/non-resident versions.

Main point. Each child calls `super.getTuition()` to obtain the base amount, then applies its own rate. The common loop remains in one place.

What to practise. Identify which version `super.getTuition()` selects and separate the common calculation from the child-specific adjustment.

Immediate-parent `super` in a three-level hierarchy

Setup. Page 7 uses `P`, `C`, and `GC`, with method `m` overridden at successive levels.

Main point. Inside `GC`, `super.m()` refers to `C`'s version. Jackie explicitly crosses out the idea of using chained `super` to jump directly to `P`.

What to practise. Before resolving `super`, mark the immediate parent of the current class.

Object visualization from the inheritance hierarchy

Setup. Pages 9–10 create `Student`, `ResidentStudent`, and `NonResidentStudent` objects.

Main point. Use the dynamic type to label the object box; then include the common parent attributes plus the appropriate child-specific attribute. Jackie stresses that the runtime pictures do not themselves reveal whether the class design used inheritance.

What to practise. Redraw the hierarchy first, then reconstruct the fields in each object box from the path.

Student expectation table

Setup. Page 11 compares references with static types `Student`, `ResidentStudent`, and `NonResidentStudent`.

Main point. Compile-time availability comes from static type. Parent members are common; each child adds only its own specialized expectation.

What to practise. Write the static type before classifying any dot-access expression as inside or outside expectation.

Smartphone ancestor-path expectations

Setup. Page 12 compares `IPhone13Pro`, `HuaweiP50Pro`, and `GalaxyS21Plus`.

Main point. Follow each class's `extends` path upward. Accumulate common methods from `SmartPhone`, branch-level methods from closer ancestors, and model-specific methods from the leaf class.

What to practise. Reconstruct the highlighted paths and expectation sets from the diagram rather than memorizing the finished annotations.

Common Pitfalls and Misconceptions

- **Giving a vague Single Choice Principle explanation.** Jackie expects a concrete maintenance argument: identify duplicated behaviour, propose a change, count the edit locations, and compare that count with the minimum possible.
- **Treating separate arrays/loops as the inheritance solution.** They belong to the no-inheritance design and are used to motivate the later single polymorphic collection; the substitution rules behind that collection are not yet taught.
- **Assuming inheritance automatically makes the design good.** Common members still belong in the parent, while child-specific members stay in the appropriate child to support the intended design principles.
- **Duplicating parent logic.** The tuition example uses `super.getTuition()` to reuse the common base calculation before applying a child-specific adjustment.
- **Confusing `this`, `super.m()`, and `super(...)`.** Jackie distinguishes current-object context, immediate-parent method access, and immediate-parent constructor invocation. The constructor call must be first, and chained `super` cannot skip to a grandparent.
- **Assuming a class can extend multiple classes.** Jackie says Java class inheritance uses one superclass; multiple interfaces are a later, separate topic.

- **Overgeneralizing the inherited-member rule.** Jackie repeatedly limits the current statement to non-private parent members and defers the detailed modifier treatment.
- **Using dynamic type or sibling members to determine expectation.** Expectation is decided at compile time from static type; members unique to a sibling subclass are outside that expectation.
- **Inferring inheritance from object boxes alone.** Jackie says the runtime diagrams for Design 2 and Design 3 can be identical. Check the class declarations for `extends`.

What You Should Be Able to Do

After this session, you should be able to:

- justify a Single Choice Principle violation by identifying duplication, proposing a change, and counting edit locations;
- explain why the no-inheritance management design grows into multiple arrays/loops and why that motivates a later polymorphic collection;
- read an `extends` declaration and identify child/subclass and parent/superclass;
- place common non-private members in the parent and specialized members in the appropriate child according to the lecture's design principles;
- distinguish inheriting a method unchanged from overriding it;
- trace the two-phase `getTuition` design using `super.getTuition()` plus a child-specific adjustment;
- distinguish `this`, `super.m()`, and `super(...)` and state the first-statement rule for the constructor call;
- resolve `super.m()` through the immediate parent and explain why chained `super` is rejected;
- reproduce Jackie's four inheritance-programming tips (1, 2A, 2B, 2C);
- draw parent/child object boxes with correct dynamic-type labels and inherited plus child-specific attributes;
- explain why runtime object structure alone does not prove inheritance;
- define expectation and classify parent-level, child-specific, and sibling-only members using static type; and
- derive a multilevel expectation by following an ancestor path upward through the smartphone hierarchy.

Suggested Practice

- Prepare for **Programming Test 2 on Wednesday, November 11** using the released guide/practice materials, solutions, Lab3 solution, and PracticeTest2 review session. Review Lab3 because it is covered by the test; its official submission deadline is also November 11.
- Continue the recent assigned exercises for later written-test/final preparation: reference versus object equality, tracing `equals`/`assertEquals`, and SMS Design Attempt 1.
- Rework pages 3–5 of Jackie's annotated iPad notes. For each design problem, explain the maintenance consequence rather than merely identifying duplicated code or multiple arrays.
- Inspect the SMS inheritance code referenced in class and map it to page 6: common parent members, inherited `register`, overridden `getTuition`, `super.getTuition()`, and `super(...)`.
- Reconstruct pages 7–8: resolve each `super.m()` through one parent level and reproduce the four design tips from memory.
- Rebuild the page-9 object diagrams and the page-11 expectation table from the hierarchy. State the static type before deciding whether a dot-access expression is inside expectation.
- Redo the page-12 smartphone exercise for the three highlighted static types by following each ancestor path and accumulating methods.
- Study the asynchronous November 11 inheritance material when it is posted. Jackie says it will build on modifiers, expectation, and substitution and warns students not to procrastinate.

Where to Review

- **Jackie's annotated iPad notes, page 2:** November 11 asynchronous-lecture reminder, Programming Test 2 resources, and current exercise priorities.
- **Jackie's annotated iPad notes, pages 3–5:** Design Attempt 2 maintenance, Single Choice Principle reasoning, separate arrays/loops, and motivation for a single polymorphic collection.
- **Jackie's annotated iPad notes, pages 6–8:**
  Design Attempt 3, class relationships, common/specialized members, overriding, parent reuse, immediate-parent lookup, and the four design tips.
- **Jackie's annotated iPad notes, pages 9–12:** runtime object visualization, the Design 2/Design 3 comparison, static-type expectation, and the `SmartPhone` ancestor-path exercise.
- **Lecture recording, roughly 0:00–0:14:** Wednesday logistics, Programming Test 2/Lab3 resources, Design Attempt 2 maintenance, and the separate-array/polymorphic-collection motivation.

- **Recording, roughly 0:14–0:55:** Design Attempt 3, `extends`, inherited and child-specific members, overriding, `super`, one-superclass clarification, four design tips, and runtime object visualization.
- **Recording, roughly 0:55–1:15:**
  Expectation definition, static-type rule, student expectation table, smartphone ancestor paths, and closing Wednesday reminders.

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 moves from a design problem to a class hierarchy, from the hierarchy to shared versus specialized code, and then from runtime object structure to compile-time expectation. If a design-principle justification, `super` resolution, or expectation decision remains unclear afterwards, return to the recording and replay that 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: justify the maintenance problem in Design Attempt 2, rebuild the three-class student inheritance hierarchy, trace both uses and the one-level scope of `super`, draw child object boxes with inherited attributes, redo the student static-type expectation table, and derive expectations from the smartphone ancestor paths. The goal is not to memorize Jackie's finished arrows or tables, but to be able to reconstruct the design, inheritance, runtime-visualization, and compile-time reasoning from the class relationships themselves.