EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 1, May 5

Course Setup, Error Models, and Variable Types

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 7

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 10

Suggested Practice 11

Where to Review 11

Closing Reminder 12

Main Ideas

- This first lecture set the course tone: how to use the syllabus, open letter, public course site, eClass site, lecture recordings, labs, tests, office hours, and classroom protocols.
- EECS2030 was framed as a demanding second-year transition course, so the first two weeks are meant to rebuild a common OOP foundation before newer topics begin.
- Labs are low-stress mastery practice: submitting an honest best attempt on time earns the lab marks, but the real goal is to internalize the material for individual tests.
- Programming and written tests are individual assessments with much stricter rules than labs; students must be able to work without collaboration, network access, AI tools, or outside resources.
- Jackie previewed the major course ideas: APIs as public method headers, JUnit 4 tests as specifications, aggregation and composition, inheritance, polymorphism, and dynamic binding.
- The lecture introduced the Java development path from editing code to compiling, running, detecting crashes, and checking expected output.
- The technical core distinguished compile-time errors, runtime exceptions, and logical errors, then connected these distinctions to debugging and JUnit-based correctness.
- The final OOP review exercise contrasted primitive fields, object-reference fields, and array-reference fields using a memory-diagram view of `int i`, `B b1`, and `B[] b2`.

Roadmap

Course tone, communication, and learning environment

Jackie began by explaining that Lecture 1 was mainly for orientation, policy, syllabus clarification, and questions; the substantial OOP review starts in the next class. Students were encouraged to interrupt with questions, use "Jackie" as the preferred form of address, and give feedback about how the course materials and iPad presentation support learning.

The lecture also set expectations for the in-person environment: avoid side conversations, keep phones away except for genuine needs, do not use headphones for unrelated media, and use laptops only for course-related notes or Java work. This matters because the course is demanding and the class needs a focused environment for both Jackie and the students.

Course sites, materials, and how to keep up

Jackie distinguished the two main course spaces. The eClass site is mainly for announcements, guides, practice questions, written tests, and course-evaluation items. The public

lecture site is where students should expect lecture recordings, slides, iPad notes, lab instructions, the syllabus, the open letter, weekly schedule, and semester calendar.

The slides are self-contained and no textbook is required, but Jackie emphasized that the annotated iPad notes are especially important because they show examples being worked from scratch. Lecture recordings are meant for catching up and improving comprehension after class, not as an invitation to avoid the in-person work.

Students who are not yet formally enrolled were told to keep attending, keep following the labs and tests, and contact Jackie with their name, student number, and Passport York ID so that eClass access can be arranged. Lab 0 Part 1 and Part 2 were available without eClass login, but later lab material will require eClass.

Lab structure, Lab 0, and the first programming test

The course has Lab 0 Part 1, Lab 0 Part 2, and Labs 1–5 as submissions. Labs count for 5% of the course and are participation-based: if students submit their best attempt before the deadline, they receive full lab marks. Jackie framed this policy as a way to reduce stress while still requiring serious practice.

The first immediate tasks are Lab 0 Part 1 and Lab 0 Part 2. Lab 0 Part 1 was announced as due May 11 at midnight, while Lab 0 Part 2 was announced as due May 20. Jackie also highlighted the written JUnit-format document and the Point Collector example, especially because primitive arrays will recur in later programming tests.

Programming Test 0 is scheduled during the May 25 lab session, covers Lab 0 Parts 1 and 2 plus the in-class OOP review, and is worth 2%. Jackie may also poll the class about optional mock-up test sessions before then. The purpose of Programming Test 0 is partly to help students experience the test format before the later, heavier tests.

Assessment rules, office hours, attendance, and accommodations

The course grading scheme is labs 5%, four programming tests 30%, two written tests 15%, and a 3-hour paper-and-pen final exam worth 50%. Written tests are eClass-based and mostly multiple choice, with possible typed questions. Programming and written tests are individual, take place in Leung 106, and do not allow teamwork, network access, AI tools, or outside resources.

For missed tests, Jackie emphasized that students must initiate communication by email by the end of the test day. Students registered for accommodations were asked to contact Jackie by the Friday of the first week so that arrangements could be discussed.

Office hours are scheduled for Monday, Tuesday, and Thursday, 7:15 p.m.–8:00 p.m., in LAS 2043, with a Zoom option. In-person students have priority. Jackie also emphasized that in-person communication after class, in labs, or during office hours is preferred over long email discussions, especially for grading, progress, or course-strategy questions.

Attendance is tracked with iClicker but is not part of the grading scheme. Students were asked to enroll in the EECS2030E iClicker course by Thursday. Attendance can only help as one possible positive factor in borderline final-grade discretion; missing lecture does not directly remove marks.

The open letter and the intended study mindset

Jackie asked students to read the open letter because it explains expectations for tests, the final exam, programming-test standards, engagement, and study habits. The key message was that the course is test-heavy and foundational, so students need steady practice rather than last-minute cramming.

The lab policy makes practice low-stress, but it also requires honest self-assessment. Students should not treat full lab credit as proof of mastery. They should ask whether they can reproduce a solution independently under test conditions. Jackie also warned that AI tools can hallucinate or fabricate, so students need enough foundational skill to evaluate AI output rather than accepting it blindly.

Course learning outcomes and the technical arc of EECS2030

Jackie previewed the course learning outcomes. The first major idea is implementing an API: the public view of a Java class, especially its method headers. A method header includes the method name, return type, and parameter or input types.

JUnit 4 test cases are central to the course. In labs and tests, students must be able to read JUnit tests to infer the required classes, methods, and behaviours. Jackie encouraged students to experiment with writing or modifying JUnit tests during practice, but clarified that graded tests and exams will expect students to read JUnit tests, not write JUnit tests from scratch.

Aggregation and composition will appear around Lab 3, including ideas such as copy constructors. The largest later topic is inheritance, including design reasons for inheritance, Java programming mechanisms such as `super` and `instanceof`, and the major concepts of polymorphism and dynamic binding. Topics such as recursion, linked lists, proofs, and running-time analysis were identified as mostly belonging to later courses such as EECS2101 and EECS3101.

Adapting to second-year programming

Jackie contrasted first-year programming experiences, such as hardware, Android, or console applications, with the more abstract second-year style. In EECS2030, problems are specified mainly through APIs and JUnit tests. Students must reason about input values, output values, object state, and correctness without relying on visual feedback.

This shift also led into regression testing: keep a set of tests, rerun them whenever the software changes, and treat any new failure as a sign that the latest change may have broken earlier behaviour. Jackie connected this mindset to professional responsibility: software and engineering work should be grounded in knowledge, competence, and evidence that the implementation behaves correctly.

Software development process: compile time versus run time

The iPad notes then focused on the software-development flowchart. Students create or edit Java classes in Eclipse, save code, compile it, run it, observe whether it crashes, and finally

check whether the output matches the expected output. Each stage can reveal a different kind of problem.

Compile-time errors happen before a program can run. Jackie separated syntax errors, such as a missing semicolon or misspelling `class`, from type errors, such as assigning a `String` value to an `int` variable or using incompatible reference types. The inheritance-related example `C1 obj = new C2();` was used to preview that assignments between different class types depend on the inheritance relationship.

Run-time problems happen after the program begins executing. A program may crash with an exception, or it may finish without crashing but still produce output that does not match the JUnit tests. Jackie called this second kind of problem a more "creative" error: the code runs, but it is still wrong.

Runtime errors and reference reasoning

The runtime-error discussion focused on `ArrayIndexOutOfBoundsException`, `NullPointerException`, and a later inheritance-related `ClassCastException`. For an expression such as `a[i]`, the invalid cases are when the index is negative or when it is at least `a.length`; the valid indices run from 0 through `a.length - 1`.

For a method call such as `obj.m(...)`, a null pointer error means the method is being invoked through a reference that does not currently refer to an object. Jackie showed the defensive idea of checking that `obj != null` before making a method call, and connected this to patterns students will see later in methods such as `equals`.

OOP review exercise: primitive values and references

The final exercise used the class fragment

```
class A {
  int i;
  B b1;
  B[] b2;
}

class B { ...  }
```

to compare what different fields can store at run time. A primitive `int` field stores an integer value directly. A `B` reference field stores a reference to a `B` object when it refers to one. A `B[]` reference field stores a reference to an array object whose elements can themselves hold `B` references.

The memory diagram is meant to build the habit of distinguishing an object from a reference to that object. This distinction will matter for arrays, method calls, aliasing, copying, null checks, and later inheritance examples.

Key Terms, Notation, and Structures

- **API**: The public view of a Java class; in this lecture, Jackie emphasized method headers as the key API information.

- **Method header**: The method name, return type, and parameter or input types; students often infer these from JUnit tests in labs and programming tests.
- **JUnit 4**: The testing framework used in this course; students must be able to read JUnit tests as specifications for required behaviour.
- **Lab best attempt**: The lab policy that gives full lab marks for an honest, on-time submission, while still expecting students to learn enough to work independently later.
- **Programming test**: An individual, resource-free assessment in the lab room where code must be developed without teamwork, network access, AI tools, or outside references.
- **Primitive array**: The required array-based structure for course implementations and programming tests; Jackie explicitly ruled out library collections such as `ArrayList` and `LinkedList` for programming tests.
- **Compile time**: The stage before running the program, when Java code is checked for issues such as syntax errors and type errors.
- **Run time**: The stage when the program is executing; this is when exceptions or wrong outputs can appear.
- **Syntax error**: A violation of Java's grammar, such as a missing semicolon or a misspelled keyword.
- **Type error**: An inconsistent manipulation of values, such as trying to store a `String` in an `int` variable or assigning incompatible reference types.
- **Runtime exception**: A crash during execution; examples from the lecture include `ArrayIndexOutOfBoundsException`, `NullPointerException`, and later `ClassCastException`.
- **Logical error**: A case where the program runs without crashing but its output does not match the expected output or JUnit tests.
- **Regression testing**: Rerunning existing tests after changes to check that new edits did not break earlier behaviour.
- **Primitive type**: A type such as `int` whose value is stored directly in the variable or field.
- **Reference type**: A type such as `B` or `B[]` whose variable or field stores a reference to an object or array object, rather than storing the whole object directly.
- **Valid array index**: For an array `a`, the valid integer indices start at 0 and end at `a.length - 1`; an index below 0 or at least `a.length` is invalid.
- **Null reference**: A reference value that does not currently refer to an object; invoking a method through it causes a `NullPointerException`.
- **Inheritance**: A later major topic in which one class extends another; Jackie previewed it as both a design topic and a Java programming topic.
- **Polymorphism and dynamic binding**: Later inheritance-related concepts that explain how reference types and runtime object types interact in method calls.

Key Examples and Demonstrations

Lab 0 and the Point Collector example

Setup: Jackie identified Lab 0 Part 1, Lab 0 Part 2, the JUnit-format document, and the Point Collector example as the first review materials students should work through.

Main point: Lab 0 is not just an easy participation task; it is the foundation for Programming Test 0 and for the primitive-array style that will recur throughout the course.

What to practise: Work through the Lab 0 videos and notes, then redo the Point Collector-style array reasoning without copying. Be able to explain how a class can use an array attribute and how JUnit tests reveal the required methods and behaviour.

Course learning outcomes and API/JUnit expectations

Setup: The iPad notes list CLOs for APIs, testing, documentation, aggregation/composition, inheritance, recursion, linked lists, proofs, and running-time analysis.

Main point: EECS2030 mainly emphasizes API implementation, reading JUnit 4 tests, aggregation/composition, and inheritance. Some listed topics are mostly left to later courses.

What to practise: Given a small JUnit test, identify the class name, constructor calls, method names, return types, parameter types, and expected output. Then write the minimal Java skeleton that would compile against the test.

Software-development flowchart

Setup: The iPad notes show a flowchart from creating or editing a Java class in Eclipse, to compiling, running, checking for crashes, and checking expected output.

Main point: Correctness is not a single checkpoint. A program must first compile, then run without the relevant exceptions, and then satisfy the expected behaviour.

What to practise: Take a small Java example and classify each failure as a syntax error, type error, runtime exception, or logical error. Fix the failure and rerun the relevant tests.

Compile-time type examples

Setup: Jackie used examples such as `int i = "2030";` and `C1 obj = new C2();` to discuss type consistency.

Main point: Type errors are not merely spelling mistakes. They arise when values are manipulated in ways that Java's type system does not allow. For class references, compatibility may depend on inheritance, which will be developed later.

What to practise: Write several short assignments involving primitives and references, predict which ones compile, and then verify in Eclipse. For inheritance-related assignments, mark them as previews rather than trying to master the full rule now.

Runtime exception conditions

Setup: Jackie analyzed array access `a[i]` and method calls such as `obj.m(...)`.

Main point: `ArrayIndexOutOfBoundsException` is caused by an invalid index, while `NullPointerException` is caused by trying to invoke a method through a reference that is `null`.

What to practise: For each array access, identify the valid index range before running the code. For each method call through a reference, ask whether the reference is guaranteed to be non-null at that point.

Variable-type memory diagram

Setup: The final OOP exercise compared `int i`, `B b1`, and `B[] b2` inside `class A`, with `class B { ... }` as the referenced object type.

Main point: Primitive fields store values directly, while reference fields store references. A reference to an array is not the whole array; it is a way to reach the array object, whose elements can then refer to other objects.

What to practise: Redraw the diagram with your own addresses such as `0x123` and `0x234`. Label what is stored directly in each field, what is an object, and what is only a reference.

Common Pitfalls and Misconceptions

Treating full lab marks as proof of mastery

Pitfall: Because labs give full marks for an honest on-time attempt, it is tempting to submit something and move on without being able to reproduce the reasoning.

Why it causes problems: Programming tests require independent work with no collaboration, network access, AI tools, or external resources. A lab submission that was completed with help does not by itself show that you can solve a similar problem under test conditions.

How to avoid it: After each lab, redo the important parts without looking. Use the solution, walkthroughs, or help only after making your own attempt and identifying exactly where you got stuck.

Confusing lab collaboration rules with test rules

Pitfall: Students may assume that because collaboration, online resources, or AI tools can be used for learning in labs, similar support is acceptable during tests.

Why it causes problems: Tests are individual assessments. Using disallowed resources during a test is both a policy problem and a sign that the lab practice has not yet become independent mastery.

How to avoid it: Practise at least some lab-style tasks in test mode: no external help, no web search, no AI tools, and no copying from previous code.

Using Java library collections when primitive arrays are required

Pitfall: Reaching for `ArrayList`, `LinkedList`, or another library collection when a programming test expects primitive-array implementation.

Why it causes problems: Jackie explicitly stated that programming tests require primitive arrays and do not allow those library collections for the relevant implementations.

How to avoid it: Study Lab 0 Part 2 and the Point Collector example carefully. Practise storing, counting, accessing, and updating elements in arrays directly.

Stopping once the code compiles

Pitfall: Assuming that a program is correct because Eclipse shows no compile-time errors.

Why it causes problems: Compiling only means there are no syntax or type errors detected by the compiler. The code may still crash at run time or produce output that fails the JUnit tests.

How to avoid it: Treat the workflow as three checks: compile, run without the relevant exceptions, and match expected outputs.

Mixing up syntax errors and type errors

Pitfall: Treating all compile-time errors as simple spelling or punctuation mistakes.

Why it causes problems: Many second-year errors are type errors, where the tokens may be spelled correctly but the values are being used inconsistently.

How to avoid it: When Eclipse reports an error, ask first whether the grammar is wrong or whether the types do not match the operation, assignment, method call, or object creation.

Ignoring boundary and null cases

Pitfall: Accessing arrays without checking the valid index range, or invoking methods without knowing whether the reference is non-null.

Why it causes problems: These habits lead directly to `ArrayIndexOutOfBoundsException` and `NullPointerException`, two runtime errors Jackie highlighted.

How to avoid it: Before using `a[i]`, check that the index is from 0 through `a.length - 1`. Before calling `obj.m(...)`, know why `obj` cannot be `null`, or guard the call appropriately.

Confusing objects with references

Pitfall: Saying that a reference variable "stores the object" or that a primitive variable such as `int i` is an object.

Why it causes problems: This confusion makes arrays, method calls, null references, aliasing, copying, and inheritance much harder to reason about.

How to avoid it: Use memory diagrams consistently: primitives store values directly; reference variables store references to objects or arrays.

Accepting AI output without judgment

Pitfall: Using AI-generated code or explanations as if they must be correct.

Why it causes problems: Jackie warned that AI tools can hallucinate or fabricate. In this course, you are responsible for judging whether code compiles, runs, and satisfies the tests.

How to avoid it: Use AI, peers, and online resources as learning aids for labs only, then verify every claim through Java rules, JUnit tests, debugger traces, and your own explanation.

What You Should Be Able to Do

After this lecture, you should be able to:

- locate the syllabus, open letter, lecture recordings, slides, iPad notes, lab instructions, weekly schedule, and semester calendar on the appropriate course sites;
- explain the difference between the public lecture site and the eClass site;
- state the main grading components: labs 5%, programming tests 30%, written tests 15%, and final exam 50%;
- explain the difference between the low-stress lab policy and the strict individual-test policy;
- identify the immediate Lab 0 tasks, the May 11 and May 20 lab deadlines announced in lecture, and the May 25 Programming Test 0 focus;
- describe what an API is in this course and identify a method name, return type, and parameter types from a method header;
- read a small JUnit 4 test case to infer what class, constructor, method, return value, or object behaviour is expected;
- explain why primitive arrays, not `ArrayList` or `LinkedList`, are required for programming-test implementations;
- distinguish compile time from run time;
- classify an error as a syntax error, type error, runtime exception, or logical error;
- state when `ArrayIndexOutOfBoundsException` occurs for an expression such as `a[i]`;
- explain why invoking `obj.m(...)` can cause `NullPointerException`;
- draw a small memory diagram distinguishing `int i`, `B b1`, and `B[] b2`;
- explain why a reference variable stores a reference rather than the whole object;
- describe how regression testing helps detect whether a new code change broke earlier behaviour;
- explain why steady weekly practice is especially important in a course with a 50% final exam.

Suggested Practice

- Read the syllabus and the open letter, then write down the policies that affect your own study plan: lab submission, missed tests, test rules, office hours, attendance, and accommodation arrangements.
- Complete Lab 0 Part 1 and begin Lab 0 Part 2 early; do not wait until the deadline to discover gaps in your OOP or array background.
- Study the JUnit-format document and the Point Collector example. For each JUnit test, identify what class or method it is specifying before looking at any solution.
- Set up Eclipse and bring it to the next lecture if possible, since Jackie planned to use the next class for live coding and a debugger demonstration.
- Install iClicker and enroll in the EECS2030E course before the Thursday trial attendance checks.
- Redraw the software-development flowchart from the iPad notes and label where syntax errors, type errors, exceptions, and logical errors appear.
- Create tiny Java examples that intentionally trigger one syntax error, one type error, one `ArrayIndexOutOfBoundsException`, and one `NullPointerException`; then fix each one.
- Redraw the `class A/class B` memory diagram using your own example addresses and explain the diagram aloud.
- Redo any Lab 0 or lecture example without AI, web search, or copied code to check whether your practice is becoming test-ready mastery.
- When using AI or online help during labs, verify the result by compiling, running JUnit tests, and explaining why the code satisfies the specification.

Where to Review

- **iPad notes, page 1**: Use the title page to remember the lecture structure: syllabus, open letter, introduction, professional ethics, software-development process, and the OOP variable-types exercise.
- **iPad notes, page 2**: Review the CLO page for the course arc: APIs, JUnit 4, aggregation/composition, inheritance, and the topics mostly left for later courses.
- **iPad notes, page 3**: Review the compile-time versus run-time flowchart and the distinction between syntax errors and type errors.
- **iPad notes, page 4**: Review the runtime-error page for array bounds, null pointers, class-cast preview, and the idea of wrong output as a logical error.

- **iPad notes, page 5**: Redraw the `int i`, `B b1`, and `B[] b2` memory diagram until the primitive/reference distinction feels natural.
- **Lecture recording**: Revisit the syllabus and course-site walkthrough for logistics, the open-letter discussion for expectations, and the final technical segment for errors and variable types.
- **Public lecture site**: Use it for the syllabus, open letter, weekly schedule, semester calendar, lecture notes, slides, recordings, Lab 0 materials, the JUnit-format document, and the Point Collector example.
- **eClass**: Use it for announcements, future test guides, practice questions, written-test activity, and the midterm course evaluation when released.
- **Lab 0 materials**: Treat Lab 0 Part 1, Lab 0 Part 2, and the primitive-array examples as preparation for Programming Test 0, not merely as participation submissions.

Closing Reminder

This digest is a roadmap for reviewing Lecture 1, not a replacement for the lecture recording, syllabus, open letter, iPad notes, slides, lab materials, or your own Java practice. The most important next step is active work: complete Lab 0 carefully, trace and redraw the diagrams, run small examples in Eclipse, use the debugger, and check whether you can reproduce the reasoning without outside help. That is what turns the low-stress lab environment into preparation for the individual tests and the final exam.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 2, May 7

OOP Review: Models, Tests, and Java Types

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 7

Common Pitfalls and Misconceptions 9

What You Should Be Able to Do 10

Suggested Practice 11

Where to Review 12

Closing Reminder 13

Main Ideas

- Lecture 2 began the OOP review by moving from course setup into the programming habits and conceptual models needed for Lab 0, later labs, and Programming Test 0.
- Jackie emphasized separation of concerns: the `model` package supplies the implementation, while the `junit_tests` package acts as the specification and client that checks expected versus actual behaviour.
- JUnit starter tests are not just grading tools; they are a way to infer required classes, constructors, method names, parameter types, return types, and observable behaviour.
- The lecture framed object orientation through the Observe–Model–Execute process: observe real-world entities, model them as compile-time classes, then instantiate and monitor runtime objects.
- A central OO distinction is that objects of the same class share the same attribute and method definitions, but each object has its own instance-specific attribute values.
- Method-call tracing depends on the context object and timing: the same method call can produce different results on different objects, or on the same object after mutation.
- Java types were reviewed as sets of allowed runtime values, with an important distinction between primitive types that store values directly and reference types that store addresses to objects.
- Jackie introduced the course's preferred arrow-and-box visualization for references and connected it to `NullPointerException`: a method call fails when its context object is `null`.

Roadmap

Announcements, Lab 0, and preparation for Programming Test 0

Jackie first reminded students that the Lecture 1 recording and notes had been posted and that the syllabus might still be updated until May 18, with explicit announcements if changes occur. Students were again told to read the open letter, because it sets expectations for how to study, practise, and interpret assessments.

The immediate priority was Lab 0 Part 1, using the tutorial videos and PDF notes, with the deadline identified as Monday, May 11. Lab 0 Part 2 was to be released on Monday, May 11 and was described as more demanding because it involves arrays in classes. Programming Test 0 was previewed for May 25 during the scheduled lab session, worth 2%, with a mock-up programming test tentatively planned for May 22.

Jackie also reminded students that office hours begin today and run Monday, Tuesday, and Thursday from 7:15 PM to 8 PM, with in-person students receiving priority over Zoom. The iClicker attendance check in this lecture was only a trial and did not count.

Review materials and the debugger habit

The review period was described as lasting through roughly Lecture 5 or possibly part of Lecture 6. Students were told not to wait passively for lecture coverage: they should work ahead through the OOP review slides, Lab 0 Part 1, Lab 0 Part 2 when released, and the written notes explaining how to derive Java code from JUnit starter tests.

Jackie strongly recommended learning to use Eclipse's debugger and breakpoints rather than relying on `System.out.println`. Printing may feel natural from earlier courses, but it does not scale well to larger programs, and debugger use will matter for EECS2030, EECS2101, EECS3101, and larger projects.

Separation of concerns: `model` versus `junit_tests`

The first technical topic was the course structure of two main packages. The `model` package contains the implementation: classes, constructors, accessors, mutators, and the logic of the Java solution. The `junit_tests` package contains starter tests, assertions, and expected-versus-actual comparisons.

Jackie described the tests as a specification of *what* the software should do, while the model is the implementation of *how* that behaviour is achieved. The relationship is one-directional: the tests use the model, but the model should not depend on the tests. The model is like a supplier, and JUnit is like a client using the supplied public methods.

The `console_apps` style was crossed out in the notes for this course. Students may have used `main`, keyboard input, and console printing in earlier courses, but EECS2030 labs and tests emphasize model classes checked through JUnit, not console applications.

Reading JUnit assertions as specifications

Jackie focused on `assertEquals(expected, actual)` as the most common assertion students will see. The first argument is the expected value supplied by the test; the second is the actual value produced by a method call from the student's model implementation. If the two runtime values match, that assertion passes; otherwise, the test fails.

A student asked whether `assertEquals` works only for certain types. Jackie explained that it is overloaded for many types, such as integers, booleans, characters, and strings. This matters because the expected value in a test often helps determine the return type that the implemented method must have.

Observe–Model–Execute

The next major diagram presented object orientation as a process. In the observe phase, we identify real-world or imagined entities and the requirements that matter for the problem. In the model phase, we turn those requirements into an object-oriented design, especially classes with attributes and methods. In the execute phase, assuming the program compiles, we instantiate templates, create objects, invoke methods, test, and debug.

The `Person` example used Jim and Jonathan to separate entities, attributes, changes, inquiries, and templates. For a BMI-related problem, weight and height are relevant attributes; gaining weight or growing taller are changes; getting weight or computing BMI

are inquiries; and `Person` is the template that later becomes a Java class. The notes also showed point objects in a two-dimensional plane as another domain where entities can be abstracted into classes.

OO concepts versus Java OOP terminology

Jackie distinguished object orientation as a design paradigm from object-oriented programming as its expression in a language such as Java, C#, C++, or Python. In this mapping, templates become classes, instances become objects, changes become mutators or setters, and inquiries become accessors or getters.

The OOP slide also introduced refactoring: changing internal workings, often private attributes or helper methods, without changing what external users observe through the public interface. Helper methods were previewed as an important way to reduce duplicated code.

Common attributes and instance-specific values

A key principle was stated in two parts. First, entities of the same kind share the same set of attributes; for example, `Person` objects may all have weight and height. Second, the values of those attributes may differ from one instance to another; Jim's weight need not equal Jonathan's weight.

Jackie then traced a small `Person` example with `jim` and `jon`. Both objects support `getWeight()` and `gainWeight(2)`, but calling the same method on different context objects can give different results. Calling `getWeight()` before and after `gainWeight(2)` also gives different results, because the object's state has changed over time.

The important tracing habit is to identify the context object to the left of the dot, update that object's state after a mutator call, and remember that identical source-code lines can behave differently when executed at different times.

Types, primitive values, and reference values

The final major topic was Java's type system. Jackie defined a declared type `T` of a variable `x` as the set of values allowed to be stored in `x` at runtime. Java is statically typed: variables must be declared with a type, and the compiler can use those declarations to catch many errors before the program runs. Python was mentioned as a contrast where variable declarations are not required in the same way.

Primitive types, such as `int`, store values directly. If `i` is declared as an integer, assigning `23` is type-consistent, while assigning a `String` or a floating-point value such as `46.7` without an appropriate cast is not.

Reference types, such as `Person`, `String`, arrays, or classes from the Java API, store addresses to objects rather than storing the full object data inside the variable. Jackie first described a lower-level address view and then introduced the more efficient arrow-and-box view that students should use for tracing.

Beginning the null-reference error discussion

The lecture ended by connecting reference variables to null-reference runtime errors. In `obj.m(...)`, `obj` is the context object and `m` is the method. An NPE occurs when that context object is `null`.

Key Terms, Notation, and Structures

- **Separation of concerns**: The design habit of keeping implementation and testing roles distinct; here, tests check the model.
- `model` **package**: The package containing the Java classes and methods that students implement.
- `junit_tests` **package**: The package containing starter tests, assertions, and expected-versus-actual checks that specify required behaviour.
- **Specification**: In this lecture, the JUnit tests describe what behaviour must be produced, even though they do not tell students every implementation step.
- **Implementation**: The Java logic placed in the model classes to satisfy the specification.
- `assertEquals(expected, actual)`: A JUnit assertion where the first value is the expected result and the second value is produced by the student's code.
- **Expected value**: The value the test says should be produced.
- **Actual value**: The runtime value returned or produced by the implemented method call.
- **Overloading**: A Java feature where a method name can have multiple versions with different parameter types; Jackie used this to explain why `assertEquals` can compare several kinds of values.
- **Observe–Model–Execute**: The OO process used in the notes: observe entities and requirements, model them as classes, then execute by creating and manipulating runtime objects.
- **Entity**: A real-world or imagined thing being modelled, such as Jim, Jonathan, or a point on a plane.
- **Attribute**: A property used to characterize an entity, such as weight, height, name, or coordinates.
- **Change**: An operation that modifies an entity's state; in Java OOP this corresponds to a mutator or setter.
- **Inquiry**: An operation that reports or computes information without changing state; in Java OOP this corresponds to an accessor or getter.

- **Template**: The OO design idea of a shared pattern for similar entities; in Java this becomes a class.
- **Class**: A compile-time Java template defining common attributes and common method definitions.
- **Object**: A runtime instance created from a class, with its own instance-specific attribute values.
- **Accessor/getter**: A method that retrieves or computes information, usually with a non-`void` return type, such as `getWeight()`.
- **Mutator/setter**: A method that changes an object's state, often with return type `void`, such as `gainWeight(2)` or a `set...` method.
- **Context object**: The object to the left of the dot in a method call; in `jim.getWeight()`, `jim` is the context object.
- **Dot notation**: Java's notation for invoking a method or accessing a member through a context, such as `obj.m(...)`.
- **Method-call chain**: A sequence such as `obj1.m1().m2().m3()`, where each call returns something on which the next call is invoked.
- **Refactoring**: Modifying internal code structure without changing the externally visible behaviour.
- **Helper method**: A usually private method used to reduce duplicated logic; Jackie previewed it as a refactoring tool.
- **Reference aliasing**: The situation where different variables store the address of the same object; Jackie previewed it as a later tracing challenge.
- **Declared type**: In a declaration such as `T x;`, `T` specifies the set of runtime values allowed for `x`.
- **Statically typed language**: A language such as Java where variables must be declared with types that the compiler checks.
- **Dynamically typed language**: A language style, mentioned with Python as contrast, where explicit variable type declarations are not required in the same way.
- **Primitive type**: A Java type such as `int`, `boolean`, or `char` whose variables store values directly.
- **Reference type**: A type such as `Person`, `String`, arrays, or API classes whose variables store references or addresses to objects.
- `null`: A special reference value meaning that the variable does not currently refer to an object.

- **Arrow-and-box diagram**: Jackie's preferred tracing visualization: a reference variable points by an arrow to an object box, or to `null` when it has no object.
- `NullPointerException`: A runtime error caused by trying to call a method on a `null` context object.

Key Examples and Demonstrations

Model versus JUnit tests

Setup: The notes show `model`, `junit_tests`, and `console_apps`, with arrows from tests to model and a large cross through the console-application style.

Main point: EECS2030 programming work is driven by tests as specifications. The model supplies classes and methods; the tests call those methods and compare expected against actual results.

What to practise: Open a small starter test and identify which package is acting as the client, which classes must be supplied in the model, and which method headers are implied by the assertions.

`assertEquals(expected, actual)`

Setup: Jackie wrote the two-argument pattern for `assertEquals`, then answered a student question about whether it applies only to certain variable types.

Main point: The first argument is the expected value; the second is the actual value from your code. The expected value also gives clues about the return type your method must have.

What to practise: For each assertion in a starter test, label the expected value, the actual expression, the called method, the context object if any, and the required return type.

Observe–Model–Execute using `Person`

Setup: The notes use Jim and Jonathan as observed entities, identify weight and height as relevant attributes, list changes such as gaining weight, and list inquiries such as getting weight or BMI.

Main point: OO design abstracts common structure from observed entities into a template, which becomes a Java class. Runtime execution then creates particular objects from that class and invokes methods on them.

What to practise: Choose a small domain such as `Person` or `Point`. Identify entities, attributes, changes, inquiries, and a likely class name before writing Java code.

Common attributes versus instance-specific values

Setup: The notes compare `Person jim = ...;` and `Person jon = ...;` with separate object boxes. Both objects have the same attribute structure, but their weight values differ.

Main point: Objects of the same class share the same attribute definitions, while the values stored in those attributes belong to each specific object.

What to practise: Draw two objects of the same class. Give them the same attribute names but different values, then explain which parts are common class-level structure and which parts are object-specific state.

Tracing `getWeight()` and `gainWeight(2)`

Setup: The trace begins with Jim at weight 72 and Jon at weight 85. Calls to `getWeight()` are followed by calls to `gainWeight(2)`, and then by later calls to `getWeight()` again.

Main point: The same method can return different values when the context object is different. The same line can also return a different value later if a previous mutator changed the object's state.

What to practise: Retrace the example in order. After every mutator call, update the appropriate object box before predicting the next accessor result.

Declared types and allowed runtime values

Setup: Jackie wrote the declaration pattern `T x;` and explained that the declared type `T` determines which runtime values can be stored in `x`.

Main point: Java's static typing helps catch type-inconsistent assignments at compile time. For an `int` variable, an integer value such as `23` is allowed, while a `String` or an uncast floating-point value is not.

What to practise: Create tiny declarations and assignments, then predict before compiling whether each one is type-consistent. Use the compiler error messages to check your prediction.

Reference variables and arrow-and-box diagrams

Setup: The notes compare a lower-level address view of `Person p;` with a higher-level diagram where `p` points either to `null` or to a `Person` object box.

Main point: A reference variable stores an address to an object, not the object's full attributes. The arrow-and-box diagram is the practical tracing tool for this course.

What to practise: Redraw `Person p;` before and after an assignment such as `p = new Person(...)`. Show when `p` has no object and when it points to a real object with attributes.

Method calls and `NullPointerException`

Setup: The final page shows a call of the form `obj.m(...)` and labels `obj` as the context object and `m` as the method being called.

Main point: A `NullPointerException` is a runtime error that happens when the context object is `null` when a method call is executed.

What to practise: For any method call in a trace, first ask: what is the context object, and does it currently point to a real object or to `null`?

Common Pitfalls and Misconceptions

Treating JUnit tests as optional checks rather than specifications

Pitfall: Students may try to implement classes based only on intuition and then use JUnit tests only at the end.

Why it causes problems: In this course, starter tests often reveal the required class names, constructor calls, method headers, return types, and expected behaviours. Ignoring them leads to code that may not even compile against the tests.

How to avoid it: Read the test first. For every assertion, identify what method is being called, on what object, with what inputs, and what result is expected.

Calling test code from the model

Pitfall: Students may blur the roles of `model` and `junit_tests`, or imagine that implementation code should depend on test classes.

Why it causes problems: The dependency goes the other way: tests use the model. If the model relies on tests, the implementation is no longer a reusable supplier of behaviour.

How to avoid it: Keep all implementation logic in the model. Use JUnit only to exercise and check the model from outside.

Using `System.out.println` as the main debugging method

Pitfall: It is tempting to insert print statements whenever something goes wrong.

Why it causes problems: Print debugging becomes messy and ineffective as programs involve more classes, more objects, and more method calls. It also distracts from learning how the program state actually changes.

How to avoid it: Practise breakpoints and the debugger in Eclipse. Step through object creation, method calls, and state changes while watching variable values.

Reversing expected and actual values in `assertEquals`

Pitfall: Students may put the actual value first and the expected value second when using `assertEquals`.

Why it causes problems: This makes test failures harder to interpret and weakens the habit of reading tests as specifications.

How to avoid it: Always label the first argument as the expected answer and the second argument as the value produced by the model.

Forgetting the context object

Pitfall: Students may focus only on the method name, such as `getWeight()`, and ignore the object to the left of the dot.

Why it causes problems: The same method can return different values for different objects, because each object has its own state.

How to avoid it: In every method call, underline or name the context object first. Then trace the method body using that object's current attribute values.

Ignoring timing in a trace

Pitfall: Students may assume that an identical line of code always produces the same result.

Why it causes problems: Mutator calls change object state. A later accessor call on the same object may return a different value from an earlier call.

How to avoid it: Trace sequentially. After every mutator, update the relevant object diagram before evaluating later calls.

Confusing reference variables with the objects themselves

Pitfall: Students may think a variable such as `Person p` directly contains all the attributes of the person.

Why it causes problems: A reference variable stores an address to an object. Confusing the reference with the object makes aliasing, mutation, and `null` much harder to reason about.

How to avoid it: Use arrow-and-box diagrams: the variable box stores a reference, and the object box stores the attributes.

Diagnosing `NullPointerException` as a syntax or type problem

Pitfall: Students may treat `NullPointerException` like a compile-time issue.

Why it causes problems: A `NullPointerException` is a runtime problem about program state. The code may compile, but the context object is `null` when the method call is executed.

How to avoid it: For a failing call `obj.m(...)`, inspect the runtime value of `obj`. Use a diagram or debugger to find when it became `null` or why it was never assigned an object.

What You Should Be Able to Do

After this lecture, you should be able to:

- identify the immediate course tasks: complete Lab 0 Part 1 by May 11, watch for Lab 0 Part 2 on May 11, prepare for the May 22 mock test if offered, and prepare for Programming Test 0 on May 25;
- explain why the OOP review period should be studied actively instead of waiting for Jackie to cover every slide in class;
- distinguish the roles of `model`, `junit_tests`, and the discouraged `console_apps` style;
- explain why JUnit tests are specifications and why the model should not depend on the tests;

- read `assertEquals(expected, actual)` and identify the expected value, actual expression, called method, and required return type;
- explain the Observe–Model–Execute process and map it to requirements, class design, and runtime monitoring;
- distinguish entities, attributes, changes, inquiries, and templates in a small problem description;
- translate OO terms into Java terms: template/class, instance/object, change/mutator, and inquiry/accessor;
- explain the difference between common attribute definitions and instance-specific attribute values;
- trace a sequence of method calls on two different objects, keeping track of the context object each time;
- explain why a mutator call changes the result of later accessor calls;
- define a declared type as a restriction on the runtime values allowed in a variable;
- distinguish Java as a statically typed language from languages where explicit variable type declarations are not required in the same way;
- classify a type as primitive or reference in common examples such as `int`, `boolean`, `Person`, `String`, and arrays;
- draw the difference between a primitive variable storing a value directly and a reference variable pointing to an object;
- use the arrow-and-box notation for `null`, object creation, and reference assignment;
- identify the context object in `obj.m(...)` and explain when that call causes an NPE;
- describe why debugger use is preferred over `System.out.println` for labs and programming tests.

Suggested Practice

- Complete Lab 0 Part 1 carefully before the May 11 deadline, using the tutorial videos and PDF notes rather than treating it as a quick submission task.
- When Lab 0 Part 2 is released, start it early because arrays inside classes were identified as more demanding and important for Programming Test 0.
- Read the written note on how to infer Java code from JUnit starter tests. For each test, write down the implied class name, constructor, method name, parameter types, return type, and expected result.

- Practise using the Eclipse debugger on a small class. Set a breakpoint before a method call, step into the method, and inspect the context object's fields.
- Redraw the separation-of-concerns diagram with the model package, the JUnit tests package, and the discouraged console-application style.
- Write a tiny `Person` class with a weight field, an accessor `getWeight()`, and a mutator such as `gainWeight(int)`. Then write or read JUnit assertions for it.
- Retrace the Jim/Jon example from the notes. Record the weight values after each line, especially before and after `gainWeight(2)`.
- Draw arrow-and-box diagrams for `Person p;`, `p = new Person(...);`, and a method call `p.getWeight()`.
- Create one example that compiles but causes `NullPointerException`; then use the debugger to identify the `null` context object.
- Complete the assigned reading from slides 16–47 and mark the parts that need clarification next week.
- Use the optional modelling pages only as extra practice: identify critical nouns and verbs in a problem description, but remember that labs and programming tests usually give stronger hints through JUnit tests.

Where to Review

- **iPad notes, page 1**: Use the title page to remember the lecture structure: OOP review, model-versus-tests separation, Observe–Model–Execute, and Java primitive-versus-reference types.
- **iPad notes, page 2**: Review the announcements for Lab 0 Part 1, the May 11 deadline, Lab 0 Part 2 release, office hours, and the iClicker trial.
- **iPad notes, page 3**: Review the separation-of-concerns diagram, especially the roles of `model`, `junit_tests`, and the discouraged `console_apps` workflow.
- **iPad notes, page 4**: Review the Observe–Model–Execute diagram and the Jim/Jonathan and point examples for entities, attributes, changes, inquiries, templates, classes, and objects.
- **iPad notes, pages 5–6**: Review the mapping from OO concepts to Java OOP terms, the distinction between classes and objects, refactoring, helper-method preview, and dot-notation chains.
- **iPad notes, pages 7–8**: Review the common-attributes versus instance-specific-values principle and the `getWeight()`/`gainWeight(2)` tracing example.

- **iPad notes, pages 9–10**: Use the optional modelling pages to practise moving from problem descriptions to classes, attributes, and methods.
- **iPad notes, page 11**: Note the assigned reading: slides 16–47.
- **iPad notes, pages 12–13**: Review the definition of declared type, Java's static typing, and the primitive-versus-reference visualization.
- **iPad notes, page 14**: Review the beginning of the `NullPointerException` discussion: `obj.m(...)` fails when the context object `obj` is `null`.
- **Lecture recording**: Revisit the early logistics if you are unsure about Lab 0, the mock test, Programming Test 0, office hours, or iClicker; revisit the middle technical segment for JUnit and Observe–Model–Execute; revisit the final segment for types and `NullPointerException`.
- **Lab 0 materials**: Use Lab 0 Part 1 and Part 2 as the immediate bridge from this OOP review to test-ready Java practice.
- **Classes and Objects slides**: Review the slides Jackie assigned and previewed, especially slides 16–47 for the next part of the review.

Closing Reminder

This digest is only a roadmap for Lecture 2. The real work is to revisit the recording and iPad notes, complete Lab 0 carefully, read the assigned slides, practise reading JUnit tests as specifications, and trace object diagrams by hand. Pay special attention to context objects, changing object state, primitive versus reference values, and `null`; these ideas will return in labs, programming tests, written tests, and later OOP topics.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 3, May 12

OOP Review: Calls, Scope, Constructors, and Assertions

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 10

Suggested Practice 11

Where to Review 12

Closing Reminder 13

Main Ideas

- Lecture 3 continued the OOP review by connecting method calls, runtime references, variable scope, constructors, and JUnit assertions into one tracing discipline.
- Jackie returned to `NullPointerException` as a runtime error caused by trying to invoke a method through a `null` context object, especially in chained method calls.
- The lecture distinguished *parameters* in method definitions from *arguments* in method calls, and emphasized that argument types must be compatible with parameter types.
- Constructor and method overloading were reviewed through the rule that overloaded versions must have distinct parameter-type lists; parameter names alone do not distinguish versions.
- Variable scope was used to separate class-level attributes and methods from method-level parameters and local variables.
- The live Eclipse demonstration showed how default constructors work, how a customized constructor removes the implicit default constructor, and why an explicit no-argument constructor may be needed.
- Variable shadowing was presented as a silent logical error, with `this` used to refer explicitly to the current object's attributes.
- JUnit assertions were reviewed as pass/fail checks, while boolean expressions evaluate to true/false; `assertSame` and `assertNotSame` compare references, not attribute values.
- The lecture closed by contrasting accessors and mutators: mutators return `void` and should be called for their side effects, while accessors return values that should normally be used.

Roadmap

Announcements, Lab 0 Part 2, and preparation for Thursday

Jackie began with immediate course logistics. Lab 0 Part 1 had already been due, and Lab 0 Part 2 had been released. Lab 0 Part 2 involves arrays, so it should be studied carefully because array-based development will matter for Lab 1 and later programming tests. The stated priority was to complete Lab 0 Part 2 by next Wednesday, May 20; Lab 1 was also to be released then. Jackie also asked students to try the posted exercise on tracing OO programs with arrays and reference aliasing before Thursday, treating it like a paper tracing exercise rather than something to type directly into Eclipse.

Returning to `NullPointerException`

The lecture resumed from the previous class's discussion of reference types and `null`. Jackie described dot notation as a runtime address lookup: in a call such as `obj.m(...)`, Java goes to the memory address stored in `obj` and invokes method `m` on that object. If the address stored in the context object is `null` or otherwise does not correspond to a valid object, the program crashes with a `NullPointerException`. The important study point is that this is not a syntax or type error; the program can compile and still fail at runtime.

Finding the source of an NPE in a method chain

Jackie then extended the idea to a chained call such as `co.m1().m2().m3()`. Even if the line compiles, there are several possible sources of a `NullPointerException`: `co` may be `null`; if not, `co.m1()` may return `null`; if not, `co.m1().m2()` may return `null`. The checks occur from left to right. Jackie emphasized a debugging habit: find the line reported by Eclipse, set a breakpoint, and use the debugger or expression panel to evaluate each part of the chain rather than guessing.

Parameters versus arguments

The notes then moved to the distinction between parameters and arguments using a `Point` example. In a method or constructor definition such as `Point(double x, double y)`, `x` and `y` are parameters. In a call such as `new Point(2.5, -3.6)`, the concrete values are arguments. Similarly, in `p1.getDistanceFrom(p2)` and `p1.move('R', 7.6)`, the values supplied in the call are arguments. Jackie highlighted this distinction because it will matter later for call-by-value, and because every method call must supply arguments whose types are compatible with the parameter types.

Constructor overloading and ambiguity

The constructor-overloading example showed a subtle compile-time problem. A `Person` class with one constructor `Person(double newWeight)` and another `Person(double newHeight)` does *not* compile, even though the parameter names are different. To the Java compiler, both have the same constructor signature `Person(double)`. A call such as `new Person(72.6)` would be ambiguous because the compiler cannot know whether the number is meant to initialize weight or height. Jackie stated the rule: if a method or constructor is overloaded, all versions must have distinct lists of parameter types.

Variable scope inside a class

The scope example separated attributes, methods, parameters, and local variables. Class-level attributes such as `i` and `j` may be accessed within methods of the same class. Method names are also class-level, so one method can call another method in the same class. Parameters such as `p1` and local variables such as `m`, however, are only available within the

method where they are declared. Jackie used small assignment examples to show the difference between type compatibility and scope validity: a line can have compatible types but still fail because the variable being used is out of scope.

Using Eclipse to test scope rules

Jackie recommended experimenting in Eclipse by creating small classes, declaring variables in different places, and checking whether red underlines appear. The goal is not to build a full JUnit test for every tiny experiment, but to use Eclipse as a learning tool for compile-time rules. One practical distinction from the discussion is that local variables must be declared before use, while the relative order of attributes and methods inside a class does not matter in the same way.

Default constructors and customized constructors

In the live Eclipse demo, Jackie created a `Person` class with attributes such as `weight` and `height`. With no constructor written explicitly, Java provides an implicit no-argument default constructor, so `new Person()` compiles and initializes numeric attributes such as `double` fields to their default value `0.0`. Jackie then added a customized constructor such as `Person(double weight, double height)`. At that point, the implicit default constructor disappeared, so calls to `new Person()` no longer compiled unless a no-argument constructor was explicitly defined. This is a major constructor rule to remember for labs and tests.

Printing objects and `toString`

The Eclipse demo also showed that printing a reference variable directly does not automatically print the object's attribute values. By default, it produces a reference-style string rather than a meaningful description of the object's state. Jackie reminded students that if they want object printing to show attribute values, they must define an appropriate `toString` method in the class.

Variable shadowing and `this`

The lecture then focused on a common constructor mistake. If a `Person` class has an attribute `weight` and the constructor also has a parameter named `weight`, then an assignment such as `weight = weight;` refers to the parameter on both sides. The attribute is shadowed and does not get initialized. This is a logical error rather than a compilation error. The supported fixes are to use `this.weight = weight;` to refer to the current object's attribute, or to avoid the name clash by using a different parameter name such as `newWeight`.

Reference tracing and JUnit assertions

Jackie traced two separate `new Person(...)` calls, such as one for `jim` and one for `jonathan`. Each `new` allocates a distinct object with a distinct address, so the variables store different

references. This supported the JUnit examples: `assertTrue(jim != jonathan)` passes because the boolean expression is true, while `assertFalse(jim == jonathan)` also passes because the boolean expression is false. Jackie emphasized the category distinction: boolean expressions evaluate to true or false; JUnit assertions pass or fail.

`assertSame`, `assertNotSame`, accessors, and mutators

Finally, Jackie introduced `assertSame` and `assertNotSame`. These assertions compare references or addresses only; they do not compare object attribute values. The lecture closed with method-use rules for accessors and mutators. A mutator such as `setWeight` returns `void`, so its call cannot be printed or assigned as a value; it should be used as a standalone statement. An accessor such as `getBMI` returns a value, so printing or assigning its result makes sense, while calling it and discarding the result compiles but is not useful.

Key Terms, Notation, and Structures

- **Context object**: the object to the left of the dot in a method call, such as `obj` in `obj.m(...)`.
- `NullPointerException`: a runtime error caused when a method call tries to use `null` as the context object.
- **Method chain**: a sequence such as `co.m1().m2().m3()`, where each call except the last must return a non-`null` object suitable for the next call.
- **Breakpoint**: a debugger marker that pauses execution so you can inspect runtime state, including reference values involved in an NPE.
- **Parameter**: a variable declared in a method or constructor definition, such as `x` in `Point(double x, double y)`.
- **Argument**: a concrete value or expression supplied in a method or constructor call, such as `2.5` in `new Point(2.5, -3.6)`.
- **Compatible type**: the type of an argument must fit the type expected by the corresponding parameter.
- **Overloading**: defining multiple constructors or methods with the same name but distinct parameter-type lists.
- **Constructor signature**: the constructor name together with the number and types of parameters; parameter names do not distinguish overloads.
- **Scope**: the part of a program where a variable or method name can be used.
- **Attribute**: a class-level variable whose scope, for this review context, covers the methods inside the class.
- **Parameter or local variable**: a method-level variable that can be used only inside the method where it is declared.

- **Default constructor**: a no-argument constructor supplied implicitly by Java only when no constructor has been explicitly defined.
- **Customized constructor**: a constructor written by the programmer, often to initialize attributes using supplied arguments.
- **Variable shadowing**: a situation where a parameter or local variable has the same name as an attribute and hides that attribute within the smaller scope.
- `this`: the keyword used to refer to the current object; for example, `this.weight` refers to the current object's `weight` attribute.
- `new`: the Java keyword that creates a new runtime object and produces a reference to it.
- `toString`: a method used to define the string representation of an object when it is printed or converted to text.
- **Boolean expression**: an expression such as `jim == jonathan` that evaluates to either true or false.
- **JUnit assertion**: a test check that either passes or fails, such as `assertTrue`, `assertFalse`, `assertSame`, or `assertNotSame`.
- `assertSame` / `assertNotSame`: assertions that compare whether two reference variables store the same address or different addresses.
- **Accessor**: a method that returns information, such as `double getBMI()`.
- **Mutator**: a method that changes object state and returns `void`, such as `void setWeight(double weight)`.

Key Examples and Demonstrations

Chained method calls and NPE sources

Setup: The notes used a chained call like `co.m1().m2().m3()`.

Main point: If an NPE occurs on this line, the possible direct sources are the context object `co`, the result of `co.m1()`, or the result of `co.m1().m2()`. The last call returning `null` is not itself an NPE unless that returned `null` is then used for another method call.

What to practise: For any chained call, mark each dot, identify the expression immediately to its left, and decide which of those expressions must be non-`null` for execution to continue.

Parameters and arguments in the `Point` example

Setup: The `Point` example included constructor and method definitions such as `Point(double x, double y)`, `getDistanceFrom(Point other)`, and `move(char direction, double units)`, together with calls such as `new Point(2.5, -3.6)`, `p1.getDistanceFrom(p2)`, and `p1.move('R', 7.6)`.

Main point: Parameters belong to definitions; arguments belong to calls. The types of the arguments must be compatible with the corresponding parameter types.

What to practise: Take each method call in the example and draw arrows from each argument to the matching parameter. Then check the parameter type and the argument type.

Ambiguous `Person(double)` constructors

Setup: The `Person` class example tried to define one constructor for weight and another constructor for height, both taking one `double` parameter.

Main point: The code does not compile because both constructors have the same parameter-type list. Names such as `newWeight` and `newHeight` help human readers but do not create distinct overloads for the compiler.

What to practise: Write the constructor signatures only, ignoring parameter names, and check whether any two are identical. If they are identical, redesign the constructors.

Scope checking in `MyClass`

Setup: The notes used a class with attributes `i` and `j`, method `m1(int p1)` with local variable `m`, and method `m2(int p2)` with local variable `n`.

Main point: Attributes are available inside methods of the class, but parameters and local variables are confined to their own method bodies. A line such as using `p1` outside `m1` fails because of scope, even if the type would otherwise match.

What to practise: Recreate the example in Eclipse and intentionally move variable uses across scopes. Use the compiler errors to test your understanding.

Default constructor and customized constructor demo

Setup: Jackie created a `Person` class with `double` attributes, instantiated objects using `new Person()`, then added a customized constructor taking weight and height.

Main point: Java supplies a no-argument default constructor only until you define some constructor yourself. After a customized constructor is added, `new Person()` requires an explicitly written `Person()` constructor if you still want that call to compile.

What to practise: Create a small class with no constructor, call `new ClassName()`, then add a one-argument constructor and observe what breaks. Add the no-argument constructor explicitly and observe what compiles again.

Variable shadowing and `this`

Setup: A constructor used parameter names such as `weight` and `height`, which matched the attribute names.

Main point: Without `this`, `weight = weight;` assigns the parameter to itself. The attribute remains unchanged. `this.weight = weight;` fixes the problem by naming the attribute explicitly.

What to practise: In Eclipse, compare the highlighting for `weight = weight;` and `this.weight = weight;`. Explain which occurrence refers to the parameter and which refers to the attribute.

Reference comparison in JUnit assertions

Setup: Two variables, `jim` and `jonathan`, were assigned from two separate `new Person(...)` calls.

Main point: Since each `new` creates a distinct object, `jim != jonathan` evaluates to true and `jim == jonathan` evaluates to false. Therefore both `assertTrue(jim != jonathan)` and `assertFalse(jim == jonathan)` pass.

What to practise: First evaluate the boolean expression inside the assertion. Only after that, decide whether the assertion method expects true or false.

Accessors and mutators

Setup: The notes used `void setWeight(double weight)` as a mutator and `double getBMI()` as an accessor.

Main point: A `void` mutator call cannot be used as a value, so printing or assigning `jim.setWeight(...)` is a compile-time error. An accessor call returns a value, so it should normally be printed, assigned, compared, or otherwise used.

What to practise: For each method call you write, look at the return type first. If it is `void`, use the call as a standalone statement; if it is non-`void`, decide where the returned value should go.

Common Pitfalls and Misconceptions

Treating `NullPointerException` as a compile-time error

Pitfall: Assuming Eclipse will always warn you before an NPE can occur.

Why it causes problems: An NPE is caused by runtime state. A line can compile because the declared types are valid, but still crash because the actual reference value is `null` when the line executes.

How to avoid it: When an NPE occurs, inspect the runtime values of the context object and intermediate method-call results using a breakpoint and the expression panel.

Looking only at the full method chain

Pitfall: Seeing `co.m1().m2().m3()` as one undivided expression.

Why it causes problems: Each dot has its own context expression, and any earlier context expression being `null` prevents the rest of the chain from executing.

How to avoid it: Break the chain into pieces: `co`, `co.m1()`, and `co.m1().m2()`. Check them left to right.

Confusing parameters with arguments

Pitfall: Calling the variables in a method definition and the values in a method call by the same name without distinction.

Why it causes problems: Later topics such as call-by-value require knowing which variables belong to the method definition and which values are supplied at the call site.

How to avoid it: Label definitions as parameters and calls as arguments whenever you trace or explain a method invocation.

Using parameter names to justify overloaded constructors

Pitfall: Thinking that `Person(double newWeight)` and `Person(double newHeight)` are different because the parameter names differ.

Why it causes problems: Java overload resolution uses parameter types and their order, not parameter names. The compiler sees both as `Person(double)`, so the class does not compile.

How to avoid it: For overloaded methods, erase the parameter names mentally and compare only the parameter-type lists.

Using variables outside their scope

Pitfall: Trying to use a method parameter or local variable in another method or at the class level.

Why it causes problems: The compiler cannot resolve a variable name outside the region where it is declared, even when the intended type would be compatible.

How to avoid it: Before using a variable, ask where it was declared. Attributes are class-level; parameters and local variables are method-level.

Assuming the default constructor is always available

Pitfall: Writing `new Person()` after adding a customized constructor and expecting it still to compile automatically.

Why it causes problems: The implicit default constructor is available only when no constructor has been explicitly written.

How to avoid it: If a no-argument constructor is needed after adding another constructor, write it explicitly.

Initializing the wrong variable because of shadowing

Pitfall: Writing `weight = weight;` inside a constructor whose parameter is also named `weight`.

Why it causes problems: Both sides refer to the parameter, so the object attribute is not changed. This is a logical error, not a compilation error.

How to avoid it: Use `this.weight = weight;` or choose a different parameter name such as `newWeight`.

Expecting printed objects to show attribute values automatically

Pitfall: Printing `jim` and expecting a readable list of `Person` attribute values.

Why it causes problems: The default object string is reference-style information, not a custom state summary.

How to avoid it: Define a suitable `toString` method when the required behaviour is to display an object's attribute values.

Confusing boolean truth with assertion success

Pitfall: Saying that an assertion is "true" or "false" instead of deciding whether it passes or fails.

Why it causes problems: `assertFalse(false)` passes, while `assertFalse(true)` fails. The boolean result and the JUnit outcome are different categories.

How to avoid it: Always use two steps: evaluate the boolean expression, then evaluate whether the assertion method accepts that value.

Using a mutator call as a value

Pitfall: Trying to print or assign the result of a `void` mutator call.

Why it causes problems: `void` means no value is returned, so the call cannot be used where a value is required.

How to avoid it: Use mutator calls as standalone statements, then call an accessor if you need to observe the updated state.

What You Should Be Able to Do

After this lecture, you should be able to:

- identify the immediate preparation tasks: complete Lab 0 Part 2 by May 20, watch for Lab 1, and attempt the arrays-and-aliasing tracing exercise before the next class;
- explain why `NullPointerException` is a runtime error, not a compile-time error;
- identify the context object in a method call such as `obj.m(...);`
- list the possible direct NPE sources in a method chain such as `co.m1().m2().m3();`

- use a breakpoint and expression evaluation to debug a suspected NPE source;
- distinguish parameters in method definitions from arguments in method calls;
- check whether method-call arguments are type-compatible with the corresponding parameters;
- explain why two overloaded constructors with the same parameter-type list do not compile;
- redesign an ambiguous overloaded constructor by changing the parameter-type list or combining versions safely;
- distinguish class-level attributes and methods from method-level parameters and local variables;
- decide whether a variable use is legal by checking its scope, not only its type;
- explain when Java supplies an implicit no-argument default constructor;
- explain why defining a customized constructor removes the implicit default constructor;
- write an explicit no-argument constructor when the default-style construction is still needed;
- explain what variable shadowing is in a constructor or method;
- use `this` to refer to the current object's attribute when a parameter has the same name;
- explain why printing an object directly may not show its attribute values unless `toString` is defined appropriately;
- draw a simple memory diagram showing two separate `new` calls producing distinct objects and distinct references;
- evaluate `assertTrue`, `assertFalse`, `assertSame`, and `assertNotSame` by separating boolean evaluation from assertion pass/fail;
- distinguish reference comparison from attribute-value comparison;
- decide whether a method call should be used as a value by checking whether its return type is `void` or non-`void`.

Suggested Practice

- Complete Lab 0 Part 2 carefully, especially the parts involving arrays, because Jackie explicitly connected them to Lab 1 and later programming tests.
- Before Thursday's class, attempt the posted exercise on tracing OO programs with arrays and reference aliasing as a paper-style tracing task.

- Recreate the `Person` constructor examples in Eclipse: start with no constructor, add a customized constructor, observe the error for `new Person()`, and then define `Person()` explicitly.
- In Eclipse, experiment with the scope example by moving uses of attributes, parameters, and local variables to legal and illegal locations.
- Create a small method chain and practise identifying which context expression would have to be `null` for an NPE to occur at each dot.
- Use the debugger, not only print statements, to pause after object creation and inspect the current values of reference variables and attributes.
- Write a small `toString` method for a class with two attributes, then compare what printing the object shows before and after the method is defined.
- For each of the assertions `assertTrue`, `assertFalse`, `assertSame`, and `assertNotSame`, make up two reference variables and predict pass/fail outcomes before running any code.
- Take a few method headers from Lab 0 or the posted notes and classify each method as an accessor or mutator based on its return type and intended effect.

Where to Review

- **Lecture recording:** Revisit the parts on `NullPointerException`, chained method calls, parameters versus arguments, scope, constructors, shadowing, JUnit assertions, and accessor/mutator usage.
- **iPad/PDF notes, page 2:** Review the announcements for Lab 0 Part 2, the May 20 deadline, the Lab 1 release plan, and the arrays-and-aliasing exercise before Thursday.
- **iPad/PDF notes, page 3:** Review the NPE diagram and the method-chain example `co.m1().m2().m3()`.
- **iPad/PDF notes, page 4:** Review parameters versus arguments using the `Point` constructor, `getDistanceFrom`, and `move` examples.
- **iPad/PDF notes, page 5:** Review the constructor-overloading example and the rule about distinct parameter-type lists.
- **iPad/PDF notes, pages 6–7:** Review variable scope, especially the difference between class-level attributes and method-level parameters/local variables.
- **iPad/PDF notes, page 8:** Review the constructor and reference-variable trace involving separate `new Person(...)` calls.
- **iPad/PDF notes, pages 9–10:** Review JUnit assertion logic for `assertTrue`, `assertFalse`, `assertSame`, and `assertNotSame`.

- **iPad/PDF notes, page 11:** Review variable shadowing and the use of `this` to refer to attributes.
- **iPad/PDF notes, page 12:** Review the accessor-versus-mutator examples and the effect of `void` return types.
- **Lab 0 Part 2 materials:** Use the tutorial videos and PDF notes to practise array-based object-oriented programming and prepare for the next lab and programming-test style.

Closing Reminder

This lecture was about small-looking Java rules that strongly affect tracing, debugging, and testing. Do not treat these rules as vocabulary only. Rewatch the relevant recording segments, redraw the diagrams from Jackie's iPad notes, test small variants in Eclipse, and practise predicting compile-time errors, runtime exceptions, assertion outcomes, and object states before checking them with the IDE. The digest is only a roadmap; the learning comes from tracing, coding, debugging, and explaining the examples yourself.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 4, May 14

OOP Review: Reference Aliasing and Array Tracing

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 12

Main Ideas

- Lecture 4 continued the OOP review by moving from low-level memory sketches to compact box-and-arrow diagrams for tracing objects, references, arrays, and aliases.
- Jackie emphasized that a reference variable stores an address, not the object itself; an arrow in a tracing diagram means "stores the address of."
- Object creation with `new` was broken into declaration, anonymous object creation, constructor execution, and assignment of the new object's address to a reference variable.
- The lecture connected `this` to the current object: inside a constructor it refers to the object being created, and inside a method call it refers to the context object on the left of the dot.
- Primitive assignment and reference assignment were compared: primitive assignment copies a value, while reference assignment copies an address and may create aliasing.
- The examples showed that == on reference variables checks whether two references store the same address, not whether two objects have the same attribute values.
- Arrays were treated as reference objects, and arrays whose elements have reference type were traced carefully to distinguish the array object from the objects referenced by its elements.
- Copying elements from one `Person[]` array into another copies addresses, so corresponding array elements may become aliases even when the two array objects are distinct.
- Jackie repeatedly emphasized precise diagram updates: mutating an object through one alias changes the shared object, while assigning a new address to a reference changes only that reference variable or array slot.
- For preparation, students were reminded to complete Lab 0 Part 2, expect a Programming Test 0 guide, and use the posted aliasing-and-arrays exercise as paper tracing practice.

Roadmap

Announcements and immediate preparation

Jackie began by reminding students that there would be no lab session on Monday, May 18 because of Victoria Day. Students were told to continue working on Lab 0 Part 2, which was due Wednesday, May 20 at midnight, and to finish it earlier if possible. Jackie also said that Programming Test 0 information would be sent soon and that Programming Test 0 would cover Lab 0 Part 1 and Lab 0 Part 2. A mock-up test was planned for Friday, May 22, with time and lab-space details to be confirmed. The review would continue into the following week, but Jackie warned students not to wait passively for lecture coverage and instead to keep reviewing the slides themselves.

Moving to compact object diagrams

The lecture then introduced a more efficient tracing notation for OO code. Instead of drawing low-level memory addresses, students should draw a rectangle for each object created with `new`. The title of the box is the object's class name, the left column lists selected relevant attributes, and the right column gives the current instance-specific values. A reference variable is drawn as an arrow pointing to the object whose address it stores. Jackie stressed that students should include only attributes relevant to the trace, since later examples may have too many fields to draw every time.

Tracing object creation with `new`

Using `Person jim = new Person(72, 1.81);`, Jackie decomposed one object-creation statement into sequential ideas. First, `Person jim` declares a reference variable that can store the address of a `Person` object. Before assignment, the reference can be thought of as storing `null`. Second, `new Person(72, 1.81)` creates a new anonymous `Person` object and runs the matching constructor. Third, the assignment stores the address of that newly created object into `jim`, so the diagram arrow for `jim` should point to the new object box.

Using `this` during construction and method calls

Jackie connected this tracing to the meaning of `this`. During constructor execution, `this` refers to the object currently being created, so assignments such as `this.weight = weight` update the new object's attributes. During a method call such as `jim.getBMI()`, `this` refers to the context object `jim`. During `jonathan.getBMI()`, the same method definition is used, but `this` now refers to `jonathan`. This is why the same method body can produce different return values or side effects depending on the object on the left of the dot.

Accessors, mutators, and context-dependent effects

The `Person` example continued with an accessor `getBMI()` and a mutator such as `gainWeightBy(double amount)`. The accessor computes using the context object's current `weight` and `height`. The mutator changes the context object's state. For example, calling `jim.gainWeightBy(3)` changes only the object pointed to by `jim`; calling the same method on `jonathan` changes a different object. The important habit is to replace `this` by the correct context object every time you step into a method.

Primitive copying versus reference copying

The notes then shifted to value copying. For primitive variables, an assignment such as `int j = i;` copies the primitive value stored in `i` into `j`. If `i` stores `3`, then `j` receives its own copy of `3`. For reference variables, an assignment such as `Point p2 = p1;` copies the address stored in `p1`; it does not copy the object's fields into a new object. After that assignment, `p1` and `p2` point to the same object and are aliases.

Reference equality and object contents

Jackie used `Point` examples to emphasize the meaning of `==`. If `p2 = p1`, then `p2 == p1` evaluates to true because both variables store the same address. If `p3 = new Point(2,3)`, then `p3` points to a distinct object, so `p3 == p1` is false even if the two objects have the same `x` and `y` values. Comparing object contents is a different issue and requires an appropriate method such as `equals`, which was not the focus of this trace.

Copying primitive array elements

Before moving to arrays of objects, Jackie traced an `int[]` example. `numbers1` was initialized with primitive values, `numbers2` was created with the same length, and a `for` loop copied each entry from `numbers1[i]` to `numbers2[i]`. The loop-counter values were identified first, then each iteration was rewritten mentally as a concrete assignment. This warm-up established the tracing routine: determine the iterations, identify the left-hand side being changed, and copy the current value from the right-hand side.

Copying reference array elements and creating aliases

The main tracing exercise used `Person` objects `alan`, `mark`, `tom`, and `jim`. The line `Person[] persons1 = {alan, mark, tom};` was expanded conceptually into creating a new array and assigning its slots to point wherever `alan`, `mark`, and `tom` point. Then `persons2` was created as a different array object with the same length. A loop copying `persons1[i]` into `persons2[i]` copied addresses, so `persons1[0]` and `persons2[0]` became aliases for the same Alan object, and similarly for the other entries.

Changing an object versus reassigning a reference

The aliasing trace then distinguished two different operations.

A call such as `persons1[0].setAge(70)` changes the Alan object itself, so the change is visible through all aliases to that object, including `alan` and `persons2[0]`. In contrast, an assignment such as `persons1[0] = jim;` changes the reference stored in the array slot `persons1[0]`. It does not move the Alan object and does not change `persons2[0]`. After that reassignment, `persons1[0]` points to Jim, while `persons2[0]` can still point to Alan.

Copying array references

The final example treated arrays themselves as reference objects. With `Person[] ps1;` and `Person[] ps2;`, both references initially store `null`, so `ps1 == null`, `ps2 == null`, and `ps1 == ps2` are all true. After two separate `new` array creations, both references are non-`null`, but `ps1 == ps2` is false because the two `new` operations create distinct array objects. After `ps2 = ps1`, both references point to the same array object. Finally, `ps1 = null` changes only `ps1`; it does not force `ps2` to become `null`. This final trace reinforces that assignment changes the left-hand side only.

Key Terms, Notation, and Structures

- **Object box**: a compact diagram element representing one runtime object created from a class.
- **Object type**: the class name written at the top of an object box, such as `Person` or `Point`.
- **Selected attributes**: the subset of fields included in a diagram because they are relevant to the current trace.
- **Instance-specific attribute values**: the current values stored in one particular object's fields.
- **Reference variable**: a variable whose value is an address of an object or `null`, such as `jim`, `p1`, or `persons1`.
- **Arrow in a diagram**: notation meaning that a variable or array slot stores the address of the object being pointed to.
- `new`: the Java keyword that creates a new object and produces a fresh reference to it.
- **Anonymous object**: an object created by `new` before its address has been assigned to a named variable or array slot.
- `this`: the current object; in a constructor it is the object being created, and in a method call it is the context object on the left of the dot.
- **Context object**: the object receiving a method call, such as `jim` in `jim.getBMI()`.
- **Accessor**: a method that returns information about an object's state, such as `getBMI()`.
- **Mutator**: a method that changes an object's state, such as `gainWeightBy(...)` or `setAge(...)`.
- **Primitive value copy**: assignment that copies an actual primitive value, as in `int j = i;`.
- **Reference value copy**: assignment that copies an address, as in `p2 = p1;` or `persons2[i] = persons1[i]`.
- **Aliasing**: the situation where two or more references store the same address and therefore name the same object.
- `==` **for references**: an address comparison that checks whether two references point to the exact same object.
- **Array object**: the runtime object representing the array container itself; an array reference such as `persons1` points to this container.
- **Array element of reference type**: a slot such as `persons1[0]` that stores an address to a `Person` object or `null`.

- `array.length`: the Java field used to determine the number of elements in an array.
- **Garbage collection**: Java's eventual cleanup of an object that can no longer be reached by any reference.

Key Examples and Demonstrations

Compact box-and-arrow object diagrams

Setup: The notes introduced a `Person` object diagram with selected fields such as age, nationality, weight, and height, together with an arrow from a reference variable.

Main point: The box represents the object, the fields inside the box represent selected state, and each arrow represents a reference storing the object's address. Multiple arrows into one box mean multiple aliases for the same object.

What to practise: Redraw the notation from memory. For each arrow, say aloud: "this variable stores the address of this object."

Breaking down `Person jim = new Person(72, 1.81)`

Setup: The object-creation line was split into declaration, `new` object creation, constructor execution, and assignment to `jim`.

Main point: A reference variable does not contain the object. The `new` expression creates the object, and the assignment stores the new object's address into the variable.

What to practise: For every `new` expression in a trace, draw a fresh object box first, then draw or update the arrow only after the assignment.

Replacing `this` by the context object

Setup: The notes used `getBMI()` and `gainWeightBy(...)` on two `Person` objects, `jim` and `jonathan`.

Main point: The same method definition is reused, but `this` changes depending on the context object. A call on `jim` uses Jim's weight and height; a call on `jonathan` uses Jonathan's state.

What to practise: Step into each method call and annotate every occurrence of `this` with the actual context object for that call.

Primitive versus reference copying

Setup: The notes compared primitive assignments such as `int j = i;` with reference assignments such as `Point p2 = p1;`.

Main point: Both assignments copy what is stored in the right-hand side variable. For primitives, that stored item is a primitive value. For references, that stored item is an address.

What to practise: Draw two side-by-side traces: one where changing a primitive variable later does not affect the other, and one where two references point to the same object after address copying.

Reference equality with `Point` objects

Setup: The `Point` trace used `p1 = new Point(2,3)`, `p2 = p1`, and `p3 = new Point(2,3)`.

Main point: `p1 == p2` is true after `p2 = p1`, but `p3 == p1` is false because `p3` was created by a separate `new`. Equal-looking field values are not the same as reference equality.

What to practise: For each == involving references, ignore the attributes first and ask only whether the arrows point to the same object box.

Copying primitive array values

Setup: `numbers1` stored values such as `1`, `2`, and `3`; `numbers2` was created with the same length and filled by a loop copying `numbers1[i]`.

Main point: For primitive arrays, each slot stores a primitive value, so the loop copies values into corresponding slots. Later changing `numbers1[0]` does not automatically change `numbers2[0]`.

What to practise: Before executing the loop body, list the loop-counter values. Then rewrite the loop body once for each value of `i`.

Copying `Person[]` element references

Setup: The main array example created `alan`, `mark`, `tom`, and `jim`, then created `persons1` from the first three and copied its elements into `persons2`.

Main point: The loop `persons2[i] = persons1[i]` copies addresses. It does not create new `Person` objects. Therefore `persons1[0]` and `persons2[0]` can point to the same Alan object, even though `persons1` and `persons2` are different array objects.

What to practise: Redraw the full trace and mark all alias paths to the Alan object after the copying loop.

Mutating through an alias versus reassigning an array slot

Setup: After copying array element references, the trace used `persons1[0].setAge(70)`, then later `persons1[0] = jim` followed by `persons1[0].setAge(75)`.

Main point: Calling `setAge` mutates the object being pointed to. Reassigning `persons1[0]` changes the address stored in that array slot. These are different actions and they change different parts of the diagram.

What to practise: For each line, decide first whether the left side is a method call on an object or an assignment to a reference slot. Then update either the object field or the arrow, not both unless the line actually requires both.

Copying array references `ps1` and `ps2`

Setup: The final trace followed `Person[] ps1;`, `Person[] ps2;`, two separate `new` array creations, `ps2 = ps1`, and `ps1 = null`.

Main point: Arrays are objects too. Two separate `new` expressions create distinct array objects. `ps2 = ps1` makes both references point to the same array object. `ps1 = null` changes only `ps1`, leaving `ps2` still pointing to the array.

What to practise: Recreate the true/false table for `ps1 == null`, `ps2 == null`, and `ps1 == ps2` at each execution point.

Common Pitfalls and Misconceptions

Thinking a reference variable stores the object itself

Pitfall: Treating `jim` or `p1` as if the variable directly contains all object attributes.

Why it causes problems: This makes aliasing impossible to reason about, because students may expect assignment to copy objects rather than addresses.

How to avoid it: Draw variables and objects separately. The variable holds an address; the object box holds fields and values.

Forgetting that each `new` creates a distinct object

Pitfall: Assuming two references are the same because the objects were constructed with the same field values.

Why it causes problems: `p3 = new Point(2,3)` creates a different object from `p1 = new Point(2,3)`, so `p3 == p1` is false even if the attributes match.

How to avoid it: Every time you see `new`, draw a fresh box unless the code is only referring to an existing object.

Using == as a content comparison for objects

Pitfall: Reading `p1 == p2` as "do these two objects have the same data?"

Why it causes problems: For references, == compares addresses. Two separate objects with identical attributes are not ==.

How to avoid it: Interpret == on references as "do these arrows point to the same box?" Use content-comparison methods only when they are explicitly part of the code or topic.

Believing reference assignment copies an object

Pitfall: Interpreting `p2 = p1` or `persons2[i] = persons1[i]` as creating a new object with copied fields.

Why it causes problems: The assignment copies an address and creates an alias, so later mutations through one reference may be visible through another.

How to avoid it: On assignment, update only the left-hand side variable or slot so that it points wherever the right-hand side currently points.

Confusing an array object with the objects stored in its elements

Pitfall: Assuming `persons1 == persons2` must be true if `persons1[0] == persons2[0]` is true.

Why it causes problems: The arrays themselves may be two distinct array objects, while corresponding slots store aliases to the same `Person` objects.

How to avoid it: Draw the array containers as separate boxes, then draw the element arrows from each slot to the referenced objects.

Mixing up mutation and reassignment

Pitfall: Treating `persons1[0].setAge(70)` and `persons1[0] = jim` as the same kind of update.

Why it causes problems: A mutator changes the object reached by the reference; assignment changes which address the reference slot stores.

How to avoid it: For a method call, update an attribute inside an object box. For an assignment, update the arrow or stored address on the left-hand side.

Thinking aliases stay synchronized as variables

Pitfall: Believing that after `ps2 = ps1`, setting `ps1 = null` also makes `ps2` become `null`.

Why it causes problems: Assignment is a one-time copy of an address. Later assignments to one variable do not automatically rewrite the other variable.

How to avoid it: Remember that the left-hand side of an assignment is the only reference variable or slot being changed.

Tracing loops without first listing the iterations

Pitfall: Jumping directly into a `for` loop over an array and losing track of which index is being updated.

Why it causes problems: A small index error changes the alias structure and leads to wrong conclusions about which objects are shared.

How to avoid it: Before updating the diagram, write the actual values of `i` and rewrite the loop body for each iteration.

What You Should Be Able to Do

After this lecture, you should be able to:

- draw a compact object diagram with a class-name title, selected attributes, current values, and reference arrows;
- explain what an arrow in an OO tracing diagram means;
- decompose an object-creation statement into reference declaration, object creation by `new`, constructor execution, and assignment of the address;

- identify what `this` refers to during a constructor call and during an ordinary method call;
- trace an accessor call by replacing `this` with the context object and reading the correct fields;
- trace a mutator call by updating the field values of the correct context object;
- distinguish primitive value copying from reference address copying;
- determine whether two reference variables are aliases of the same object;
- evaluate == comparisons between reference variables by checking whether the arrows point to the same object;
- trace an array of primitive values and explain why copying primitive elements does not create shared objects;
- trace an array of reference values and explain how copying elements can create aliases;
- distinguish the array object from the objects referenced by its elements;
- identify all alias paths to a particular object in a diagram;
- decide whether a line of code mutates an object or reassigns a reference variable or array slot;
- complete the true/false table for `ps1 == null`, `ps2 == null`, and `ps1 == ps2` across declaration, creation, aliasing, and null assignment;
- prepare for Programming Test 0 by reviewing Lab 0 Part 1, Lab 0 Part 2, and the tracing ideas used in these examples.

Suggested Practice

- Complete Lab 0 Part 2 before the May 20 deadline, paying special attention to arrays and reference-type behaviour.
- Redraw the compact object-diagram notation from page 2 of the iPad notes and explain each component in words.
- Retrace the `Person jim = new Person(72, 1.81)` example and explicitly label declaration, `new`, constructor execution, and assignment.
- For each `getBMI()` or `gainWeightBy(...)` call in the notes, write which object `this` refers to before computing or updating anything.
- Rework the primitive-versus-reference copying example using your own small numbers and your own `Point` objects.

- Redo the `Person[]` aliasing example on paper without Eclipse; after each line, update only the arrows or fields that actually change.
- In the array-aliasing example, list all alias paths to Alan at each important point in the trace.
- Recreate the final `ps1`/`ps2` table from memory, then check it against the iPad notes.
- For any `for` loop over an array, first list the loop-counter values, then rewrite the loop body once per iteration before updating the diagram.
- Use the posted tracing exercise on arrays and reference aliasing as a paper-test rehearsal for upcoming assessments.

Where to Review

- **iPad/PDF notes, page 1:** lecture topic map for value copying, reference aliasing with arrays, and copying arrays.
- **iPad/PDF notes, page 2:** compact object visualization notation: object box, selected attributes, instance-specific values, and reference arrows.
- **iPad/PDF notes, pages 3–4:** object creation, `this`, accessors, mutators, and context-object tracing for `Person` objects.
- **iPad/PDF notes, page 5:** primitive versus reference copying with `int` variables and `Point` references.
- **iPad/PDF notes, page 6:** copying primitive array values and tracing a `for` loop over array indices.
- **iPad/PDF notes, pages 7–8:** the main `Person[]` aliasing trace and the question about all alias paths to Alan.
- **iPad/PDF notes, page 9:** copying array references, including the true/false table for `ps1`, `ps2`, and `null`.
- **Lecture recording:** revisit the explanation of why `persons1 == persons2` may be false while `persons1[0] == persons2[0]` is true.
- **Lab 0 Part 2 materials:** review the array-related development carefully, since Jackie connected it directly to Lab 1 and Programming Test 0 preparation.
- **Posted exercise:** attempt the tracing exercise on OO programs with arrays and reference aliasing before checking the solution walkthrough.

Closing Reminder

This digest is a roadmap for Lecture 4, not a substitute for doing the tracing yourself. The central skill from this lecture is precision: draw the object boxes, draw the arrows, update the left-hand side of assignments, update object fields after mutator calls, and check reference equality by looking at addresses rather than attribute values. Revisit Jackie's iPad notes, the lecture recording, Lab 0 Part 2, and the posted aliasing exercise until you can reproduce the diagrams without relying on the completed trace.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 5, May 19

Using `this`, Static Variables, and Automatic IDs

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 8

Where to Review 9

Closing Reminder 9

Main Ideas

- Lecture 5 continued the OOP review by connecting reference aliasing, the `this` keyword, and class design decisions that affect runtime object state.
- The `Person` marriage example showed how a method should be designed from a concrete trace: identify the context object, the parameter object, the precondition, and the postcondition before writing the general method body.
- Jackie emphasized that `this` is not only for resolving variable shadowing; inside an instance method, it represents the current context object and can itself be assigned into another reference field.
- Boolean conditions were reviewed through the legal-versus-illegal marriage check, including conjunction, disjunction, negation, short-circuit evaluation as a preview, and De Morgan's laws.
- Static and non-static variables were contrasted: non-static attributes belong to individual objects, while a static variable is a single class-level value shared by all instances.
- The `Counter` example introduced the tracing habit of handling static variables at "step zero" and drawing them separately from object boxes.
- The `Account` ID example showed why manually passing IDs through a constructor is fragile and how a static counter can automatically generate unique sequential IDs.
- Assessment preparation was a major theme: Programming Test 0 was scheduled for Monday, May 25, the optional mock-up test for Friday, May 22, and Lab 0 Part 2 remained the immediate priority.

Roadmap

Administrative reminders and immediate priorities

Jackie reminded students that the notes template had been posted, office hour would run from 7:15 PM to 8 PM, and Programming Test 0 would take place on Monday, May 25 during enrolled lab sessions. The guide for Programming Test 0 had been released, the optional mock-up test was scheduled for Friday, May 22 from 16:00 to 16:50, and the main short-term work was to finish `Lab0P2`. Jackie also explained that Lab 1 would introduce the format students should expect later: starting from JUnit tests and building the model package from scratch.

The `Person` marriage example: using `this` beyond shadowing

The notes then moved to the example `Person jim = new Person("Jim"); Person elsa = new Person("Elsa"); jim.marry(elsa);`. The key design question was how to implement `public void marry(Person other)` when `Person` has a `name` field and a `Person spouse`

field. Because `spouse` has reference type, a newly created person has `spouse == null` unless it is explicitly assigned. Jackie used this example to show that `this` represents the context object `jim`, while `other` represents the argument `elsa`; concrete names such as `jim` and `elsa` should not be written directly inside the general method body.

Preconditions, postconditions, and bidirectional reference updates

The marriage operation was organized around a precondition and a postcondition. Before the update, the marriage is legal only when both objects are currently single: `this.spouse == null && other.spouse == null`. After a successful update, the references must be symmetric: `this.spouse = other;` and `other.spouse = this;`. Jackie also highlighted the timing issue: an expression such as `jim.spouse.spouse` causes a `NullPointerException` before the marriage, but after the two assignments it traces back to `jim`.

Boolean operators and De Morgan's laws

The lecture briefly reviewed conjunction, disjunction, and negation, using both mathematical notation and Java notation. In Java, conjunction is written `&&`, disjunction is written `||`, and negation is written `!`. Jackie then gave an exercise: rewrite `marry` so that the `if` condition detects an illegal marriage rather than a legal one. One safe approach is to wrap the legal condition in a single negation; another is to apply De Morgan's law, such as $\neg(P \wedge Q) \equiv (\neg P) \vee (\neg Q)$. The point was not just syntax, but preserving the same behaviour while flipping the branch structure.

Manual account IDs and why they are fragile

The second major topic was static versus non-static state, introduced through an `Account` class. In the manual design, the constructor took both an owner and an ID, and the caller had to supply an ID that did not clash with any existing account. Jackie pointed out two drawbacks: the caller must remember all previous IDs, and duplicate IDs still compile because two identical integers are not a type error. The duplicate-ID problem is therefore a logical design problem, not something the compiler can prevent automatically.

Static versus non-static variables

The `Counter` example separated a non-static variable `L` from a static variable `g`. A non-static field is instance-specific: each object has its own copy. A static variable is class-level: all objects share one copy. Jackie emphasized two practical conventions: initialize non-static attributes in constructors, and initialize static variables where they are declared. Accessing a static variable through an object reference, such as `c1.g`, may compile, but it causes an Eclipse warning and hides the correct mental model; the preferred form is `Counter.g`.

Tracing static state with "step zero"

For tracing code that contains static variables, Jackie introduced a special habit: before tracing object creation, first draw and initialize the static variable separately. In the `Counter`

trace, `g` started at 0 before any `Counter` object existed. The objects `c1` and `c2` then each received their own copy of `L`. Calling `incrementLocal()` changed only the context object's `L`, whereas calling `incrementGlobal()` or updating `Counter.g` changed the one shared static value.

Automatic account IDs with a static counter

The account-design problem was then revisited. Instead of requiring the caller to pass an ID, the improved `Account` class used a static `globalCounter`, initialized to 1, and a non-static `id`. In the constructor, the current counter value is copied into `this.id`; then the counter is incremented for the next account. Thus the first account gets ID 1, the second gets ID 2, and the shared counter is ready for the next object. Jackie also mentioned a string-ID variation, such as concatenating a prefix like `"ID"` with the numerical value.

Key Terms, Notation, and Structures

- **Context object**: The object on the left of the dot in a method call; inside the method body, it is represented by `this`.
- **Recursive type**: A class that has a field of its own type, such as `private Person spouse;` inside the `Person` class.
- **Precondition**: A condition that should hold before a method performs its main action; in the marriage example, both `spouse` fields should be `null`.
- **Postcondition**: A condition that should hold after the method finishes; in the marriage example, the two `spouse` references should point to each other.
- **Aliasing**: The situation where multiple reference variables or fields store the address of the same object.
- **Non-static variable / attribute**: A per-object field; each object has its own copy and its own instance-specific value.
- **Static variable**: A class-level variable shared by all instances of the class; it should be accessed through the class name, such as `Counter.g`.
- **Step zero for tracing**: The habit of drawing and initializing static variables before tracing any object creation or instance method calls.
- **Manual ID management**: A design where callers pass IDs into constructors and are responsible for avoiding duplicates.
- **Automatic ID management**: A design where the class uses a static counter to assign unique IDs internally.
- **Logical operators**: Java uses `&&` for AND, `||` for OR, and `!` for NOT; Jackie previewed short-circuit evaluation and used these operators in the marriage legality condition.

- **De Morgan's laws**: Logical equivalences such as $\neg(P \wedge Q) \equiv (\neg P) \vee (\neg Q)$ and $\neg(P \vee Q) \equiv (\neg P) \wedge (\neg Q)$, useful when flipping a condition.
- **Warning versus error**: Some code, such as accessing a static variable through an instance reference, may compile but still indicate a misunderstanding through an IDE warning.

Key Examples and Demonstrations

Implementing `marry(Person other)`

Setup: The `Person` class has a `name` field and a `Person spouse` field. The example creates `jim` and `elsa`, then calls `jim.marry(elsa)`.

Main point: Use the concrete example to derive the general method body. In the call `jim.marry(elsa)`, `this` corresponds to `jim`, and `other` corresponds to `elsa`. A legal marriage requires both `spouse` references to be `null`; a successful update requires both directions of the relationship to be assigned.

What to practise: Draw the two `Person` boxes before and after the method call. Then write the method using `this` and `other`, not the concrete names from the example.

Tracing `jim.spouse.spouse`

Setup: The expression `jim.spouse.spouse` is evaluated before and after the marriage updates.

Main point: Timing matters. Before marriage, `jim.spouse` is `null`, so the second dot causes a `NullPointerException`. After the symmetric assignments, `jim.spouse` refers to `elsa`, and `jim.spouse.spouse` traces back to `jim`.

What to practise: Trace the expression by following one dot at a time. Stop immediately if an intermediate reference is `null`.

Flipping the marriage condition

Setup: The original structure checks whether the marriage is legal and performs the two assignments in the `if` branch. Jackie then asked students to rewrite it so that the `if` branch detects the illegal case.

Main point: The condition can be inverted either by writing a top-level negation around the original legal condition or by using De Morgan's laws to distribute negation over the conjunction. This is a reasoning exercise about preserving behaviour while changing control flow.

What to practise: Let P mean `this.spouse == null` and Q mean `other.spouse == null`. Rewrite `!(P && Q)` into the equivalent OR-based form and test both versions with small examples.

The `Counter` trace: local versus global state

Setup: The `Counter` class has a non-static `L` and a static `g`. The tester creates two `Counter` objects and calls local and global increment methods.

Main point: Each object has its own `L`, but all objects share one `g`. The static variable must be drawn separately and initialized before object tracing begins. In the lecture trace, local increments affected only the chosen object, while global increments updated the shared value.

What to practise: Redraw the trace with a separate static box marked for `g`. For each line, say whether the line changes `c1.L`, `c2.L`, or `Counter.g`.

Manual versus automatic `Account` IDs

Setup: The manual version of `Account` takes an explicit ID in the constructor. The automatic version removes that parameter and uses a static `globalCounter`.

Main point: Manual IDs rely on the caller to avoid duplicates, and duplicate IDs still compile. The static-counter version stores the current counter value into the new object's `id`, then increments the counter for the next object.

What to practise: Implement the automatic version from scratch. Then create several accounts and trace the sequence of `id` values and the changing `globalCounter`.

Common Pitfalls and Misconceptions

Writing concrete variable names inside a general method

Pitfall: Writing names such as `jim` or `elsa` inside `marry(Person other)`.

Why it causes problems: Those names belong to a particular example or tester scope; they are not available inside the method body. The code will not generalize and may not compile.

How to avoid it: Translate the example into method-scope names: use `this` for the context object and `other` for the argument object.

Forgetting the second assignment in a bidirectional relationship

Pitfall: Implementing only `this.spouse = other;` and forgetting `other.spouse = this;`.

Why it causes problems: The relationship becomes inconsistent: one object may point to the other while the reverse direction still says `null`.

How to avoid it: State the postcondition before coding. For marriage, both directions must be true after a successful call.

Dereferencing before checking timing and nullness

Pitfall: Assuming `jim.spouse.spouse` is always meaningful.

Why it causes problems: Before `jim.spouse` has been assigned to another object, it is `null`; the next dot causes a `NullPointerException`.

How to avoid it: Trace dot expressions one step at a time and check whether each intermediate reference is `null` before continuing.

Negating an AND condition incorrectly

Pitfall: Treating `!(P && Q)` as if it were equivalent to `!P && !Q`.

Why it causes problems: That changes the logic. It would only detect the case where both parts fail, not the case where either part fails.

How to avoid it: Use De Morgan's law: $\neg(P \wedge Q)$ becomes $\neg P \vee \neg Q$.

Thinking duplicate IDs are a compile-time problem

Pitfall: Assuming Java will prevent two accounts from receiving the same integer ID.

Why it causes problems: The compiler checks types, not uniqueness of runtime values. Passing `1` twice is still type-correct.

How to avoid it: Move the responsibility into the class design by using a static counter or another internal uniqueness mechanism.

Drawing static fields inside every object

Pitfall: Putting `g` or `globalCounter` inside each object box as if it were an instance attribute.

Why it causes problems: This hides the fact that there is only one shared value. It will lead to wrong traces when one object changes the static variable and another object observes the change.

How to avoid it: Draw static variables separately before object tracing begins, and mark them as class-level shared state.

Accessing static variables through objects as a habit

Pitfall: Using `c1.g` or `this.g` for a static variable.

Why it causes problems: It may compile, but Eclipse warns because the notation suggests that `g` belongs to a specific object.

How to avoid it: Use the class name: `Counter.g` or `Account.globalCounter`.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain why `Person spouse` is a reference-type field and why its default value is `null`;
- implement a method such as `marry(Person other)` using `this` and the parameter name rather than concrete tester variables;
- state a precondition and postcondition for a mutator that changes references;

- trace expressions such as `jim.spouse` and `jim.spouse.spouse` before and after reference assignments;
- distinguish boolean expression values from the control-flow role they play inside an `if` statement;
- apply negation and De Morgan's laws to rewrite a legal-condition check into an illegal-condition check;
- distinguish static variables from non-static attributes in ownership, lifetime, initialization, access, and tracing;
- draw static variables separately from object boxes and initialize them before tracing object creation;
- trace how local object state and shared static state change through methods calls such as `incrementLocal()` and `incrementGlobal()`;
- explain why manual account ID assignment is fragile and why duplicate IDs are logical errors rather than compilation errors;
- implement an automatic ID generator using a static counter and a non-static `id`;
- prepare for Programming Test 0 by finishing `Lab0P2`, reading the guide, and practising from scratch rather than only rereading solutions.

Suggested Practice

- Finish `Lab0P2`, especially the parts involving primitive arrays inside Java classes, and make sure you can redo the ideas without copying.
- Read the Programming Test 0 guide and use the optional mock-up test to practise importing, exporting, and managing time in the lab-test setting.
- Redraw the `Person` marriage diagram from the notes, then implement `marry(Person other)` from the diagram.
- Write two versions of the marriage legality check: one checking legal first, and one checking illegal first. Confirm that the two versions behave the same on representative cases.
- In Eclipse, test a few boolean expressions involving `&&`, `||`, and `!`; then rewrite them by De Morgan's laws and compare outputs.
- Import or recreate the static-versus-non-static `Counter` code and trace it before running it.
- Modify the `Account` class so that it uses `globalCounter`; create several accounts and predict each ID before running the code.

- Try the posted exercises Jackie mentioned: tracing reference-type return values, anonymous objects, and helper methods. Use the walkthrough videos only after attempting them yourself.

Where to Review

- **iPad/PDF notes, page 1:** Use the title page to locate the lecture's three main technical themes: the marriage example, static versus non-static variables, and manual versus automatic account IDs.
- **iPad/PDF notes, page 2:** Review the immediate logistics: Programming Test 0 on Monday, May 25, optional mock-up test on Friday, May 22, and the priority to finish `Lab0P2`.
- **iPad/PDF notes, page 3:** Revisit the `Person` and `marry(Person other)` diagram, especially the translation from `jim/elsa` to `this/other`.
- **iPad/PDF notes, pages 4–5:** Review the logical operators and De Morgan exercise for flipping the marriage condition.
- **iPad/PDF notes, page 6:** Review why manual ID management in `Account` creates a burden on the caller and does not prevent duplicate IDs at compile time.
- **iPad/PDF notes, page 7:** Study the `Counter` diagram for static versus non-static tracing, including the separate static box and the warning about accessing static state through an instance.
- **iPad/PDF notes, page 8:** Review the automatic account ID design using a static `globalCounter` and a non-static `id`.
- **Lecture recording:** Rewatch the segments where Jackie derives `marry(Person other)` from the concrete example, traces the `Counter` program, and refactors manual IDs into automatic IDs.
- **Course site materials:** Use the posted code for static versus non-static variables, the exercise PDF on the `Person`/marriage example, the Lab 0 materials, and the Programming Test 0 guide.

Closing Reminder

This digest is a roadmap for review, not a replacement for the lecture, the iPad/PDF notes, the posted code, the Lab 0 materials, or the Programming Test 0 guide. The most important work after this lecture is active: draw the reference diagrams, trace the static state separately, rewrite the boolean conditions yourself, and implement the examples from scratch before checking solutions or recordings.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 6, May 21

Caller-Callee Tracing and Static Context Errors

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- Lecture 6 marked the transition from the OOP review into exceptions: Jackie treated caller/callee reasoning and call-stack tracing as the foundation needed before Java exception handling.
- Each method call defines one caller-callee relation; to identify the relation precisely, use the enclosing method as the caller and the declared type of the context object to identify the callee.
- A method is not permanently a caller or permanently a callee; the same method can play different roles in different call expressions, including recursive and non-recursive contexts.
- A chain such as `m1()` → `m2()` → `m3()` should be traced using a call stack: callers suspend, callees execute, completed methods are removed, and execution resumes where the caller left off.
- Jackie assigned slides 5–9 on error handling via console messages so that students can explain why `System.out.println` is not enough as a serious error-handling mechanism.
- The remaining OOP review focused on static misuse: making per-object counters or arrays static creates shared state, skipped array slots, logical errors, and possible `NullPointerExceptions`.
- The `Bank.getInfo()` example showed the common compile-time error "non-static variable used in a static context" and connected it to the rule that static methods do not have a `this` context object.
- Programming Test 0 preparation was emphasized: students were told to read the test guide, practise Eclipse import/export and zip submission, use Practice Test 0, and start Lab 1.

Roadmap

Wrapping up the OOP review and starting exceptions

Jackie began by saying that the main OOP review was complete and that students should catch up independently using the slides, iPad notes, labs, and recordings. The new topic was exceptions, but the lecture deliberately started with prerequisite execution-flow concepts: caller versus callee, a chain of method calls, and the call stack. Jackie also previewed the catch-or-specify requirement as a coming topic and assigned slides 5–9 for self-study on why console messages are a weak way to signal errors.

Caller versus callee

The first technical focus was the caller-callee relation. A caller is the method that initiates a method call; a callee is the method being called. Jackie emphasized that each method call

defines exactly one such relation. For a call like `o.m2(...)` inside `C1.m1`, if `o` is declared with type `C2`, the caller is `C1.m1` and the callee is `C2.m2`. The class-method notation here is identification notation, not a claim that `m2` is static.

A method can change roles across contexts

Jackie then showed that caller/callee roles are not fixed properties of a method. The same method can be a caller in one call expression and a callee in another. Recursion is one example, but Jackie also gave non-recursive patterns: another method in the same class can call `this.m1()`, or a method in another class can create a `C1` object and call `o.m1()`. Students were assigned the analogous exercise of creating contexts that make `C2.m2` a caller rather than only a callee.

Tracing a method-call chain using a stack

The lecture then moved to a simple chain `m1()` → `m2()` → `m3()`. Jackie introduced the call stack as the runtime structure for tracking active method calls. When `m1` begins, `m1` is placed on the stack. When `m1` calls `m2`, `m1` is suspended, `m2` is placed on top, and execution continues inside `m2`. The same pattern occurs when `m2` calls `m3`. When `m3` terminates, it is removed from the stack; then `m2` resumes after the call to `m3`; when `m2` terminates, `m1` resumes after the call to `m2`. Jackie stressed that this suspension-and-resumption discipline will be needed for exceptions.

Programming Test 0 and Practice Test 0 logistics

After the call-stack discussion, Jackie reviewed immediate assessment preparation. Programming Test 0 was scheduled for Monday, May 25: students should attend their enrolled lab section, with the 1:00 PM and 2:30 PM sections distinguished, and students with university-registered accommodations directed to the 1:00 PM session. The Programming Test 0 guide was posted under Test Guides. Practice Test 0 instructions, starter code, and solution were released, and an optional mock-up session was scheduled for Friday, May 22, 16:00–16:50 in Lassonde Building 1006. Jackie emphasized practising the operational workflow: download/import the starter project into Eclipse, export the project as a zip file, and submit through the provided link.

Misusing static for per-object counters

The next technical section returned briefly to static versus non-static variables. Jackie considered a `Client` class with an `accounts` array and a `numberOfAccounts` counter. In the intended Lab 0 Part 2 pattern, the counter records how many accounts that particular client has stored, so it must be non-static. Jackie then traced what happens if `numberOfAccounts` is mistakenly declared static: `bill.addAccount(acc1)` stores in Bill's array at index 0 and increments the shared counter; `steve.addAccount(acc2)` then stores in Steve's array at index 1, skipping index 0. The skipped slot is not automatically an error, but dereferencing it later can lead to a `NullPointerException`.

Misusing static for arrays

Jackie also discussed the related mistake of declaring the `accounts` array itself as static. In that case, all `Client` objects would share the same array, so actions by Bill and Steve would modify one common storage structure instead of each client owning a separate account list. This is a logical design error for the intended model, even if some version of the code can compile.

Static context errors in the `Bank` example

The final technical example involved a `Bank` class with a non-static `branchName`, a static `nextAccountNumber`, and a static `getInfo()` method.

Jackie pointed out that using `this.branchName` inside a static method causes a compile-time error because there is no context object in a static method call such as `Bank.getInfo()`. One possible fix is to remove the non-static attribute access from the static method. Another possible fix is to make `branchName` static, but Jackie warned that this is poor design when different bank objects should have different branch names. A further design lesson was that a method should be static only when it does not require instance-specific information, such as a utility method operating only on its parameters.

Key Terms, Notation, and Structures

- **Caller**: The method that initiates a method call.
- **Callee**: The method being invoked by a caller.
- **Caller-callee relation**: The relation defined by one specific method call; identify it per call expression, not once per method name.
- **Class-method identification**: Notation such as `C1.m1` or `C2.m2` identifies the class and method involved; in this lecture it is not static-method syntax.
- **Context object**: The object to the left of the dot in a non-static method call, such as `o` in `o.m2(...)`.
- **Declared type**: The type written in the variable declaration; Jackie used it to identify the class where the callee method is declared.
- **Entry point of execution**: The starting method for a run, such as `main` or the beginning of a JUnit test case.
- **Call stack**: A runtime stack of active method calls; new calls are pushed on top, and completed calls are removed from the top.
- **Suspension and resumption**: When a caller invokes a callee, the caller waits at the call site; after the callee terminates, the caller resumes immediately after that call.

- **Catch-or-specify requirement**: A coming exception-handling rule that Jackie previewed; it depends on caller/callee and call-chain reasoning.
- **Static variable**: A class-level variable shared by all instances of the class.
- **Non-static variable**: An instance-level field; each object has its own copy.
- **Static method**: A method callable through the class name; it has no `this` context object.
- **Static context**: A context, such as a static method body, where instance fields or instance methods cannot be accessed through `this`.
- **Utility method**: A typical static method whose result depends only on its parameters, not on instance-specific fields.
- `numberOfAccounts`: The counter in the `Client` example; it should be non-static when it tracks the number of accounts stored for one client object.
- `accounts`: The array in the `Client` example; it should be non-static when each client owns its own array.
- `branchName` and `nextAccountNumber`: The `Bank` example fields used to distinguish instance-specific branch information from shared account-number information.
- **String concatenation**: Java can concatenate a `String` and an `int`, producing a new `String`, such as `"Lecture Room" + 23`.

Key Examples and Demonstrations

Identifying caller and callee in `o.m2(...)`

Setup: Inside `C1.m1`, the code calls `o.m2(...)` and `o` is declared with type `C2`.

Main point: The caller is `C1.m1`, because that is where the call is initiated. The callee is `C2.m2`, because the method name is `m2` and the declared type of the context object is `C2`. Jackie stressed that method name alone is not precise enough.

What to practise: For each method call in a small code fragment, write down the caller and callee as class-method pairs. Always locate the context object's declared type before naming the callee.

Making a method act as a callee in another context

Setup: `C1.m1` is first seen as a caller. Jackie then sketched contexts where it becomes a callee: another method in `C1` can call `this.m1()`, or another class can create a `C1` object and call `o.m1()`.

Main point: Caller/callee roles depend on a particular call site. A method can be a caller in one call expression and a callee in another.

What to practise: Complete Jackie's exercise: write two small contexts in which `C2.m2` becomes a caller. Use one same-class context and one external-class context if possible.

Tracing `m1()` → `m2()` → `m3()` with a call stack

Setup: `m1` begins execution, calls `m2`; `m2` calls `m3`; `m3` has no further method call.

Main point: A caller does not continue while its callee is running. The active calls stack up as `m1`, then `m2`, then `m3`. When `m3` terminates, it is removed and `m2` resumes; when `m2` terminates, `m1` resumes. This exact control-flow pattern is the foundation for later exception propagation.

What to practise: Redraw the three-method trace from scratch. Label the line sequence, the stack after each call, the point where each caller suspends, and the point where each caller resumes.

Tracing a mistaken static `numberOfAccounts`

Setup: A `Client` object has an `accounts` array and a counter `numberOfAccounts`. The counter is mistakenly declared static. Bill and Steve are two distinct clients, and `addAccount` stores into `accounts[numberOfAccounts]` before incrementing the counter.

Main point: Since the counter is static, Bill and Steve share the same counter. After Bill stores at index 0, Steve's first account may be stored at index 1, leaving Steve's index 0 as `null`. This is a logical error, and later dereferencing a skipped `null` slot can cause a `NullPointerException`.

What to practise: Trace Bill, Steve, and a possible third client such as Ian. Draw each client's separate array, draw the single shared counter separately, and mark skipped indices.

Tracing a mistaken static `accounts` array

Setup: Jackie asked what would happen if the `accounts` array itself were declared static.

Main point: All clients would share one array. This contradicts the intended model where each client has its own accounts. The example reinforces that "static" is a design choice about shared state, not a shortcut.

What to practise: Modify the example in Eclipse or on paper by making `accounts` static. Predict how Bill and Steve's additions interfere, then compare your prediction against the code behaviour.

Diagnosing the static `Bank.getInfo()` error

Setup: In the `Bank` class, attribute `branchName` is non-static, attribute `nextAccountNumber` is static, and method `getInfo()` is declared static. The static method attempts to use/access the non-static attribute via `this.branchName`.

Main point: A static method has no `this`. Therefore, accessing `this.branchName` in `Bank.getInfo()` gives a compile-time error. Removing the non-static access can fix the compile error; making `branchName` static also compiles, but Jackie warned that this changes the model so all bank objects share the same branch name.

What to practise: Try three fixes: remove the instance-field access, make `getInfo()` non-static, and make `branchName` static. For each one, decide whether it merely compiles or also preserves the intended design.

Common Pitfalls and Misconceptions

Using only the method name to identify caller or callee

Pitfall: Saying only "`m2` is the callee" without identifying the class where `m2` is declared.

Why it causes problems: Different classes can declare methods with the same name, so the method name alone may be ambiguous.

How to avoid it: Write caller and callee as class-method pairs such as `C1.m1` and `C2.m2`; use the declared type of the context object to identify the callee class.

Confusing lecture notation with static syntax

Pitfall: Treating notation like `C2.m2` in Jackie's explanation as meaning that `m2` must be a static method.

Why it causes problems: In this context, the dot notation was only a naming convention for identifying the class and method, not Java syntax for a static call.

How to avoid it: Separate explanation notation from executable Java syntax. Ask: "Is this a piece of code, or is Jackie naming the method's class and method?"

Forgetting that callers suspend during callee execution

Pitfall: Continuing to trace lines in the caller immediately after a method call before the callee has completed.

Why it causes problems: The runtime transfers control to the callee. The caller resumes only after the callee returns or terminates through exception behaviour to be studied next.

How to avoid it: Use the call stack. Push the callee when the call starts; do not resume the caller until the callee has been removed from the stack.

Making a per-object counter static

Pitfall: Declaring `numberOfAccounts` as static when it is meant to count the accounts stored for one specific client.

Why it causes problems: All client objects share the counter, so one client's update changes the index used by another client. This can skip array slots and later produce `null` dereferences.

How to avoid it: Ask whether the value belongs to one object or to the whole class. A counter paired with one object's array should normally be non-static.

Making an instance-owned array static

Pitfall: Declaring `accounts` as static when each client should own a separate account list.

Why it causes problems: All clients modify the same array, so the object model no longer represents separate client account lists.

How to avoid it: Keep instance-owned data non-static. Use static only when the shared state is intentional and matches the model.

Accessing instance state inside a static method

Pitfall: Writing code such as `this.branchName` inside a static method like `Bank.getInfo()`.

Why it causes problems: A static method is called without a context object, so there is no `this`. The compiler reports that a non-static variable is being used in a static context.

How to avoid it: Either make the method non-static, pass in a needed object, or restrict the static method to static data and parameter-based computation.

Fixing compilation by damaging the model

Pitfall: Making a field static only because it removes a compile-time error.

Why it causes problems: The code may compile but the meaning changes. For example, making `branchName` static forces all `Bank` objects to share the same branch name.

How to avoid it: After every compile-fix, ask whether the resulting state should be shared by all objects or remain instance-specific.

Neglecting test-time tooling practice

Pitfall: Knowing the programming concepts but losing test time because importing, exporting, zipping, or submitting the Eclipse project is unfamiliar.

Why it causes problems: Programming Test 0 includes the same operational workflow from Lab 0 and the Practice Test 0 materials.

How to avoid it: Practise the exact workflow before the test: download starter code, import into Eclipse, export the project as a zip file, and submit through the correct link.

What You Should Be Able to Do

After this lecture, you should be able to:

- Explain the difference between a caller and a callee in a specific method call.
- Identify caller and callee as class-method pairs, using the declared type of the context object when naming the callee.
- Give examples where the same method acts as a caller in one context and a callee in another.
- Trace a nested call chain such as `m1()` → `m2()` → `m3()` using a call stack.
- Explain what it means for a caller to be suspended and then resumed after the callee terminates.
- State why call-stack tracing is needed before studying exception propagation and catch-or-specify requirements.
- Explain why console messages such as `System.out.println` are not enough as a robust error-handling strategy, after studying slides 5–9.

- Analyze whether a field should be static or non-static based on whether the state belongs to the class as a whole or to each object.
- Trace the logical error caused by making a per-object counter such as `numberOfAccounts` static.
- Predict when skipped `null` array slots can lead to a `NullPointerException`.
- Explain why an instance-owned array such as `accounts` should not be static in the `Client` example.
- Diagnose the compile-time error caused by accessing a non-static field inside a static method.
- Compare multiple fixes for a static-context error and judge whether each fix preserves the intended design.
- Decide when a method is reasonably static, especially for utility-style methods that depend only on parameters.
- Locate and use the Programming Test 0 guide, Practice Test 0 materials, and Eclipse import/export workflow before the test.

Suggested Practice

- Study slides 5–9 on error handling via console messages and write down the drawbacks of using only `System.out.println` to signal errors.
- Redraw the caller-callee example from the notes and identify the caller and callee for each call using class-method notation.
- Complete Jackie's exercise: create method-call contexts that make `C2.m2` a caller, not merely a callee.
- Retrace the `m1()` → `m2()` → `m3()` example with a call stack, marking every push, suspension, termination, pop, and resumption.
- Reimplement the `Client` account-array example with `numberOfAccounts` non-static, then change it to static and predict the incorrect behaviour before running it.
- Create a test that exposes the skipped-slot problem by attempting to use an array element that remains `null`.
- Repeat the experiment by making `accounts` static and explain why all clients now share one array.
- Recreate the `Bank.getInfo()` example and try the alternative fixes Jackie discussed; record which fixes are good design and which merely remove the compile error.

- Practise Practice Test 0 under realistic timing, or attend the Friday, May 22 mock-up test if possible.
- Before Programming Test 0, practise downloading starter code, importing into Eclipse, exporting the completed project as a zip file, and submitting it through the provided link.
- Start Lab 1 and use it to practise reading JUnit tests and building the model package from scratch.

Where to Review

- **iPad/PDF notes, page 1:** Lecture 6 topic list: caller versus callee, tracing method-call chains with a stack, and catch-or-specify as the next exception-handling direction.
- **iPad/PDF notes, page 2:** Announcements: Programming Test 0 on Monday, May 25; optional mock-up test on Friday, May 22, 16:00–16:50; Practice Test 0 instructions and solution released; Lab 1 as the priority.
- **iPad/PDF notes, page 7:** Assigned reading: slides 5–9 on error handling via console messages.
- **iPad/PDF notes, page 8:** Caller-callee definitions, the `C1`/`C2` example, and the reminder that one method can be caller or callee depending on context.
- **iPad/PDF notes, page 9:** The call-stack diagram for `m1()`, `m2()`, and `m3`; use this page when retracing suspension and resumption.
- **iPad/PDF notes, page 3:** Misuse of static variables in the `Client` account-array example; review the skipped-slot trace.
- **iPad/PDF notes, pages 4–6:** Static context error in `Bank.getInfo()`, possible fixes, and the guideline for when a method should be static.
- **Lecture recording:** Revisit the caller/callee identification segment, the call-stack tracing segment, and the later static-misuse/static-context examples.
- **Course website:** Check Test Guides for Programming Test 0, Exercises for Practice Test 0 materials, and Lab 1 materials for the next implementation task.

Closing Reminder

This digest is only a roadmap for Lecture 6. To prepare properly, revisit the iPad notes and recording, work through the caller/callee and call-stack examples by hand, study the assigned slides on console-message error handling, and practise the static/non-static examples by modifying and testing code yourself. For Programming Test 0, make sure your Eclipse import/export/submission workflow is smooth before the test; reading this digest alone is not enough preparation.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 7, May 26

Handling Exceptions: Flow, Call Stacks, Catch-or-Specify Requirement

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 11

Main Ideas

- The lecture began by consolidating the **call stack** model: stack operations happen at the top, and active method calls terminate in last-in, first-out order.
- Java exceptions were introduced as disruptive events that stop ordinary execution and force a method chain to obey the **catch-or-specify requirement**.
- Jackie distinguished two handling choices: **catch** an exception to close the loop locally, or **specify** it with `throws` to propagate responsibility to the caller.
- The central programming example used a chain `Tester.main` → `B.mb` → `A.ma`, with a custom `NegativeValueException` thrown when `i < 0`.
- The lecture compared three designs: catch in `B.mb`, specify from `B.mb` and catch in `Tester.main`, or specify all the way out of `main`, which causes the JVM to terminate the run and print a stack trace.
- Students were reminded that Lab 1 is due Wednesday, June 3, that Programming Test 1 is on Monday, June 8, and that the guide and practice test for Programming Test 1 will be released later.

Roadmap

1. Course reminders and current priorities

Jackie opened with the transition after Programming Test 0. The TA will begin grading soon, and feedback will normally consist of a numerical mark plus a detailed test report showing which test cases passed or failed. Students do not automatically receive their submitted code back, but they may use a regrading request or office hours to review a specific submission.

The main study priority is now Lab 1, due Wednesday, June 3. Jackie emphasized that students should start early, struggle through the test cases before looking for solutions, and ask questions as needed. Programming Test 1 is scheduled for Monday, June 8; the policy will be similar to Programming Test 0, and a guide plus practice questions will be released about a week in advance.

2. Reviewing the call stack

The lecture returned to the call-stack trace from the previous class. A stack has a **bottom** and a **top**; all meaningful stack operations happen at the top. Jackie accepted the everyday terms "add" and "remove", but introduced the standard CS terms **push** and **pop**. A method call pushes a new frame; a terminated method is popped.

The key reading direction depends on the question. Reading bottom-to-top gives the chronological order in which methods were invoked, such as `m1`, then `m2`, then `m3`. Reading top-to-bottom gives the chronological order in which methods terminate: `m3`, then `m2`, then `m1`. This is the last-in, first-out property, and Jackie connected it directly to how exceptions will move through method-call chains.

3. Why exceptions need the stack model

Jackie introduced exceptions as special events that cannot simply be ignored. Unlike printing an error message to the console, an exception disrupts ordinary flow and forces the program to take an exception-handling path. Students were pointed back to slide 10 for the formal definition, and to the earlier assigned reading on why console messages are not enough.

The first conceptual diagram used a chain such as `main` → `m1` → `m2` → `m3`. If an exception originates in `m3`, the call stack stops growing there. The exception then has to be dealt with while moving back toward the callers. This is why caller-callee identification and stack tracing are not optional background: they are the mechanism behind exception propagation.

4. Catch-or-specify: two choices at each relevant method

Jackie described the catch-or-specify requirement using the same stack diagram. Informally, to **specify** means to propagate the exception to the caller by declaring it in the method header with `throws`. To **catch** means to handle it locally using a `try/catch` block, closing the loop so that higher callers no longer have to mention that exception.

At the method where an exception is thrown, the usual starting point in this lecture is to specify it. For example, if `m3` throws an exception, `m3` specifies it and passes responsibility to `m2`. Then `m2` has a design choice: catch it or specify it again. If `m2` specifies, responsibility moves to `m1`. If `m1` catches, callers above `m1`, such as `main`, are not subject to catch-or-specify for that exception.

5. Syntax for the catch option

The catch option uses a `try/catch` block. The method call that may throw an exception is placed inside the `try` block. The `catch` block declares the exception type and an exception variable, in the same spirit as declaring a parameter. Jackie also mentioned that a single `try` block may be followed by multiple `catch` blocks if different exception types may be thrown.

The important study point is not to memorize only the surface syntax, but to connect it to the execution model. A catch block is the place where a propagated exception is handled. If the catch happens in a method, then the method's callers do not need to catch or specify that particular exception, unless another call or another exception type creates a new requirement.

6. Syntax for the specify option

The specify option uses the keyword `throws` in the method header, after the parameter list and before the method body. For example, a method header may be extended as `public void ma(int i) throws NegativeValueException`. If a method may throw several exception types, the method header can list them after `throws`.

Jackie emphasized the spelling and placement distinction between `throw` and `throws`. Use `throw` inside a method body to create and signal an exception object at runtime. Use `throws` in the method header to declare that the method may pass that exception to its caller.

7. Custom exceptions and domain-specific error reporting

A student asked whether an exception such as `InvalidTransactionException` is built into Java. Jackie explained that many exceptions used in the course will be custom, domain-specific exceptions. For Lab 2, students will practise exception handling in a domain-specific setting, and they should expect to create exception classes relevant to the problem.

Jackie also warned against treating general runtime exceptions as good API declarations. For example, students should not design APIs by declaring `throws NullPointerException` or `throws ArrayIndexOutOfBoundsException`. Instead, define meaningful exceptions that reflect the domain rule being violated. In lecture demonstrations, console printing may be used to make traces visible, but students should avoid relying on `System.out.println` in labs and programming tests unless explicitly permitted.

8. Eclipse demonstration: `A.ma` throws `NegativeValueException`

The live coding example began with class `A` and method `ma(int i)`. If `i < 0`, the method should signal an error by a statement like "`throw new NegativeValueException("i value" + i);`". The exception object is created anonymously and thrown immediately.

Eclipse first complained that `NegativeValueException` was not yet a defined type. Jackie used Eclipse's class-generation workflow to create the exception class, then added a constructor `NegativeValueException(String s)` that calls `super(s)`. After the exception class existed, the compiler then reported that the thrown checked exception was unhandled; the fix at the origin method was to add `throws NegativeValueException` to the `ma` method header.

9. Version 1: `B.mb` catches the exception

Next Jackie created class `B` with method `mb(int i)`, which creates an `A` object and calls `ma(i)`. Because `A.ma` declares `throws NegativeValueException`, the call inside `B.mb` is subject to catch-or-specify. In Version 1, `B.mb` chooses to catch the exception locally.

This design wraps the call to `A.ma(i)` (which specifies `NegativeValueException`) inside a `try` block and catches `NegativeValueException` in `B.mb`. Since the exception is caught in `B.mb`, `Tester.main` can call `B.mb` without a red underline for this exception. From the public header of `B.mb`, there is no propagated `NegativeValueException` for `main` to handle.

10. Version 2 and Version 3: propagating responsibility

In Version 2, Jackie changed `B.mb` so that it specifies `throws NegativeValueException` instead of catching. This makes `B.mb` compile, but it moves the catch-or-specify requirement to the caller, `Tester.main`. One reasonable design is then for `Tester.main` to use a `try/catch` block around the call to `B.mb`.

In Version 3, Jackie demonstrated what happens when `Tester.main` also specifies the exception instead of catching it. If a negative input triggers the exception, the exception escapes the program's entry point, the JVM terminates the run, and a stack trace is printed to the console. Jackie called this a design to avoid for ordinary user-facing applications: the

error should be handled before it reaches a user who cannot do anything meaningful with the raw stack trace.

Key Terms, Notation, and Structures

- **Call stack**: the stack of active method calls; the most recently invoked active method is at the top.
- **Top of stack**: the only position where stack operations happen; the topmost method is the current focus of execution.
- **Bottom of stack**: the earliest active method in the current chain, often the program's entry point such as `main`.
- **Push**: add a new method frame to the top of the stack when a method is invoked.
- **Pop**: remove the top method frame when that method terminates.
- **LIFO**: last in, first out; the method invoked most recently must finish before its caller can resume.
- **Invocation order**: the bottom-to-top order in which methods were called.
- **Termination order**: the top-to-bottom order in which active methods complete.
- **Exception**: a special event represented by an exception object that disrupts normal execution and forces handling.
- **Exception propagation**: the process of moving exception responsibility from a callee to its caller when the exception is not caught locally.
- **Catch-or-specify requirement**: for checked exceptions, a method must either catch the exception or declare that it may throw it.
- **Catch**: handle an exception locally using a `try/catch` block.
- **Specify**: declare in the method header, using `throws`, that the method may pass the exception to its caller.
- `throw`: the keyword used inside a method body to signal an exception at runtime, such as `throw new NegativeValueException(...)`.
- `throws`: the keyword used in a method header to declare possible exception propagation, such as `throws NegativeValueException`.
- `try` **block**: the code region containing a call or computation that may throw an exception.
- `catch` **block**: the handler for a specific exception type; it declares an exception variable such as `NegativeValueException nve`.

- **Custom exception**: an exception class defined for the problem domain (e.g., circles, banks), such as `NegativeValueException` or `InvalidTransactionException`.
- `super(s)`: the constructor call used in the custom exception class to pass an error message to the superclass constructor.
- **Normal termination**: a method finishes its ordinary flow, for example by reaching the end or returning normally.
- **Abnormal termination**: a method stops because an exception is thrown.
- **Stack trace**: the console output showing the chain of calls when an exception reaches the JVM uncaught.
- `Tester.main` → `B.mb` → `A.ma`: the main lecture example for tracing where an exception originates and where it is caught or specified.

Key Examples and Demonstrations

Call stack: invocation order vs. termination order

Setup: The notes show a stack containing `m1` at the bottom, `m2` above it, and `m3` at the top, continuing the previous lecture's chain `m1` calls `m2`, then `m2` calls `m3`.

Main point: The same stack supports two readings. Bottom-to-top gives method invocation order; top-to-bottom gives method termination order. This is the call-stack form of last-in, first-out.

What to practise: Redraw the stack and explain why `m3` must terminate before `m2` resumes, and why `m2` must terminate before `m1` resumes.

Exception originates at the top of a call stack

Setup: The lecture used a chain `main` → `m1` → `m2` → `m3`, with an exception originating in `m3`.

Main point: Once the exception occurs, the stack does not keep growing as if execution were normal. Responsibility moves back along the caller chain until some method catches the exception, or until the exception escapes the entry point.

What to practise: Mark each edge as caller/callee, then trace where the exception responsibility sits after `m3` specifies it, after `m2` specifies it, and after `m1` catches it.

Catch syntax with one or more handlers

Setup: Jackie sketched `try` followed by one or more `catch` blocks,
such as catching `NegativeRadiusException` or catching two different exception types from the same `try` block.

Main point: The `try` block contains code that may throw; each `catch` block handles a particular exception type and can refer to the caught exception variable.

What to practise: Write a small `try/catch` skeleton by hand. Then add a second `catch` block and explain when each block would run.

Specify syntax in a method header

Setup: The notes use examples such as a `withdraw` method that specifies an exception by including "`throws InvalidTransactionException`" as part of its method header.

Main point: `throws` is not handling. It is a declaration that the method may pass the exception to its caller. The caller must then catch or specify.

What to practise: Take a method that calls a throwing method and produce two versions: one with `try/catch`, and one with a `throws` clause.

Creating and throwing `NegativeValueException`

Setup: In class `A`, method `ma(int i)` throws `NegativeValueException` when `i < 0`. Eclipse first reports that the exception type does not exist, and then reports an unhandled-exception error after the class is created.

Main point: Creating the custom exception class and throwing the exception are not enough. Because the exception is checked, `A.ma` must also specify it by including "`throws NegativeValueException`" as part of its method header.

What to practise: Recreate the example in Eclipse: write the `throw new ...` line, generate the exception class, add the String constructor with `super(s)`, and then add the `throws` clause to the method header.

Version 1: `B.mb` catches

Setup: `B.mb(int i)` creates an `A` object and calls `A.ma(i)`. In Version 1, `B.mb` wraps that call in `try/catch`.

Main point: Because `B.mb` catches `NegativeValueException`, `Tester.main` can call `B.mb` without being subject to catch-or-specify for that exception.

What to practise: Trace both cases: `i >= 0`, where the call completes normally, and `i < 0`, where the catch block runs inside `B.mb`.

Version 2 and Version 3: moving or losing control

Setup: In Version 2, `B.mb` specifies `throws NegativeValueException`, so `Tester.main` must catch or specify. In Version 3, `Tester.main` also specifies.

Main point: Specifying keeps moving responsibility upward. If the entry point also specifies and no method catches, the JVM receives the exception and the user sees a stack trace rather than a controlled program response.

What to practise: Run the three versions with a positive and a negative value. For each run, write down where the exception is thrown, whether it is caught, and what the console shows.

Common Pitfalls and Misconceptions

Treating stack operations like arbitrary list operations

Pitfall: Thinking you can remove any method from the middle of the call stack.

Why it causes problems: Stack tracing depends on top-only operations. If you remove frames out of order, you lose the logic of suspension, resumption, and exception propagation.

How to avoid it: Use push for method invocation and pop only for the current top method when it terminates.

Reading the stack in the wrong direction

Pitfall: Confusing invocation order with termination order.

Why it causes problems: For `m1`, `m2`, `m3`, the first invoked method is not the first one that terminates. This mistake leads to incorrect traces of normal calls and exceptions.

How to avoid it: Read bottom-to-top for invocation order and top-to-bottom for termination order.

Thinking an exception is just a printed error message

Pitfall: Treating `System.out.println` or `System.err.println` as equivalent to exception handling.

Why it causes problems: Printing does not force the program to stop, propagate, or handle the error. Exceptions create a formal control-flow obligation.

How to avoid it: Use printing only when Jackie uses it for tracing in lecture. For real exception handling, use `throw`, `throws`, and `try/catch`.

Confusing `throw` and `throws`

Pitfall: Using the two keywords interchangeably.

Why it causes problems: `throw` belongs inside the method body and signals an exception at runtime. `throws` belongs in the method header and declares possible propagation to the caller.

How to avoid it: Associate `throw` with "create and signal now" and `throws` with "warn the caller in the header."

Believing specify means the exception is handled

Pitfall: Adding `throws NegativeValueException` and thinking the problem is finished.

Why it causes problems: Specifying only moves the catch-or-specify requirement to the caller. The next caller may now show the compilation error.

How to avoid it: Follow the call chain upward. Keep asking: at this method, is the exception caught, or merely specified again?

Letting exceptions escape `main`

Pitfall: Fixing compilation by adding `throws` all the way to `main` and stopping there.

Why it causes problems: If the exception occurs, the run terminates and the JVM prints a stack trace. Jackie emphasized that this is not a good user-facing design.

How to avoid it: Catch exceptions at an appropriate level before they escape the entry point.

Using generic runtime exceptions as domain contracts

Pitfall: Designing an API with declarations such as `throws NullPointerException` or `throws ArrayIndexOutOfBoundsException`.

Why it causes problems: These usually indicate programming mistakes, not clean domain-level failures. They do not communicate the intended rule of the model.

How to avoid it: Create meaningful custom exceptions for domain rules, such as invalid values or invalid transactions.

Relying too much on Eclipse quick fixes

Pitfall: Letting Eclipse generate `try/catch` or `throws` without understanding why the compiler asks for it.

Why it causes problems: The code may compile, but you may not know whether the design catches at the right level or merely pushes responsibility elsewhere.

How to avoid it: Use quick fixes for repetitive setup when allowed, but write and trace simple `try/catch` blocks manually until the syntax and control flow are comfortable.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain top, bottom, push, pop, and LIFO for a method-call stack;
- read a call stack bottom-to-top for invocation order and top-to-bottom for termination order;
- trace how normal method calls suspend and resume through a chain of callers and callees;
- explain why an exception stops ordinary flow and must be handled rather than merely printed;
- describe the catch-or-specify requirement in terms of caller and callee responsibility;
- distinguish the catch option from the specify option in both syntax and control-flow effect;
- use `throw` inside a method body and `throws` in a method header correctly;
- write a simple `try/catch` block for a method call that may throw a checked exception;

- define a custom exception class with a String constructor and `super(s);`
- trace the three-class example `Tester.main` → `B.mb` → `A.ma`;
- compare Version 1, Version 2, and Version 3 in terms of where `NegativeValueException` is caught or specified;
- explain why propagating a checked exception out of `main` leads to program termination and a stack trace;
- identify why domain-specific custom exceptions are preferable to declaring general run-time exceptions in APIs;
- plan study time for Lab 1, due Wednesday, June 3, and begin preparing for Programming Test 1 on Monday, June 8.

Suggested Practice

- Redraw the stack from the notes with `m1`, `m2`, and `m3`; label invocation order, termination order, top, bottom, push, and pop.
- Revisit the exception call-stack diagram and explain in words what happens when `m3` specifies, `m2` specifies, and `m1` catches.
- Write a minimal `try/catch` skeleton by hand, then add a second catch block for a second exception type.
- In Eclipse, recreate the `A.ma(int i)` example: throw `NegativeValueException` when `i < 0`, generate the exception class, and add the required `throws` clause.
- Implement `B.mb(int i)` in two ways: one version that catches `NegativeValueException`, and one version that specifies it.
- Implement `Tester.main` for both versions and observe where the compiler places the catch-or-specify requirement.
- Run the example with a positive value and a negative value; write down which printouts or stack traces occur and why.
- Modify Version 3 so that the exception is caught before escaping `main`; compare the user experience before and after the change.
- Practise creating a custom exception with an informative message such as `"i value" + i`; explain why Java treats the whole expression as a string.
- Work on Lab 1 now, before the solution is available, and use office hours or after-class questions when a test case or design point remains unclear.

Where to Review

- **iPad/PDF notes, page 1:** lecture title and focus: exceptions, catch-or-specify, and the three handling versions.
- **iPad/PDF notes, page 2:** reminders for Lab 1, Programming Test 1, and office hours.
- **iPad/PDF notes, page 3:** stack terminology, push/pop, top/bottom, LIFO, and the `m1`/`m2`/`m3` call chain.
- **iPad/PDF notes, page 4:** what happens when an exception is thrown in a call stack, including catch-or-specify and the idea of closing the loop.
- **iPad/PDF notes, page 5:** Java syntax for `try/catch`, multiple catch blocks, and `throws` in a method header.
- **iPad/PDF notes, page 6:** the `A`, `B`, and `Tester` example, including the call stack `Tester.main` → `B.mb` → `A.ma`.
- **iPad/PDF notes, page 7:** Version 1, where `B.mb` catches `NegativeValueException`; use the two questions on the page as a self-check.
- **Lecture recording:** review the live Eclipse demonstration where `NegativeValueException` is created, specified in `A.ma`, caught or specified in `B.mb`, and either handled or propagated by `Tester.main`.
- **Posted code under exceptions:** import the three versions into Eclipse and compare their compiler requirements and runtime behavior.
- **Earlier assigned slides 5–9:** review the limitations of console-message error handling before relying on formal exceptions.
- **Lab 1 and upcoming Lab 2:** use Lab 1 as the immediate coding priority, and expect Lab 2 to require catch-or-specify and custom exception practice.

Closing Reminder

This digest is a roadmap for reviewing the lecture, not a substitute for doing the work. To learn this material, you need to redraw the stack diagrams, trace the normal and abnormal execution paths, write the `try/catch` and `throws` versions yourself, run the examples in Eclipse, and connect the syntax back to the caller-callee chain. Revisit Jackie's notes, the recording, the posted code versions, and the lab instructions while you practise.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 8, May 28

Tracing Normal versus Abnormal Flows

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- Lecture 8 continued the exceptions unit by focusing on how checked exceptions move through a call chain and where the catch-or-specify requirement is closed.
- Jackie compared three versions of the same `Tester.main` → `B.mb` → `A.ma` setup: `B.mb` catches, `B.mb` specifies and `Tester.main` catches, and both specify so the exception reaches the console.
- A major goal was to trace *normal* execution, where no exception is thrown, against *abnormal* execution, where a `throw` interrupts the current method and transfers control to a matching `catch` or to the caller.
- The lecture used both paper tracing and Eclipse debugging to show how breakpoints, the call stack, and the value of a condition such as $i < 0$ confirm the predicted execution path.
- The Bank/Account and multiple-catch examples reinforced two habits: identify the object to the left of the dot before deciding which method is called, and remember that only the first matching catch block executes.
- The main preparation message was practical: continue Lab 1, practise tracing exception flows before the next lecture, and be ready to use these ideas when exceptions are combined with JUnit testing.

Roadmap

Announcements and study priorities

Jackie began by reminding the class that Programming Test 1 is scheduled for Monday, June 8, with a guide and practice test to be released. Lab 1 is due next Wednesday, June 3, and completing Lab 1 was described as the best preparation for the upcoming programming test. Jackie also said that a solution walkthrough video would be posted after the Lab 1 deadline, so students should first try to work through the lab and its test cases themselves.

Recapping the source of the exception

The technical discussion resumed from the previous lecture's class `A`. In `A.ma`, a negative input value for `i` signals an error by throwing a custom checked exception such as `NegativeValueException`. Because the exception originates there, `A.ma` declares it with `throws`. This recap set up the recurring question for the rest of the lecture: once a callee declares a checked exception, what must each caller do?

Version 1: `B.mb` catches the exception

In Version 1, `A.ma` specifies the exception, so the call `oa.ma(i)` inside `B.mb` is subject to catch-or-specify. `B.mb` chooses the catch option by wrapping the call in a `try-catch` block.

Since the exception is caught in `B.mb`, the later call from `Tester.main` to `B.mb` is not subject to catch-or-specify for that exception. Jackie described this as closing the exception loop at `B.mb`.

Version 2 and Version 3: propagating farther

In Version 2, `B.mb` does not catch the exception; instead, it specifies the exception by declaring as part of its method header: `throws NegativeValueException`. This shifts the catch-or-specify responsibility to `Tester.main`, which catches the exception in this version. In Version 3, `Tester.main` also specifies the exception, so there is no catch inside the application chain and the exception can reach the console. Jackie emphasized that this is programmatically possible but usually poor design for user-facing code, because the user sees a raw exception rather than a controlled response.

Tracing normal execution in Version 1

Jackie next traced Version 1 with a non-negative value such as `i = 10` or `i = 5`. The flow enters `Tester.main`, calls `B.mb`, then calls `A.ma`. Since $i < 0$ is false, `A.ma` follows the normal branch, returns to `B.mb`, and `B.mb` continues after the call inside the `try` block. Because no exception occurred, the catch block is skipped, `B.mb` returns, and `Tester.main` resumes after the call. The key study point is not the exact print statements, but the return pattern: no exception means the `try` block completes normally.

Tracing abnormal execution in Version 1

Jackie then retraced Version 1 with a negative value such as `i = -5` or `i = -10`. The early steps are the same until `A.ma` evaluates $i < 0$ as true. At that point `A.ma` throws the exception and terminates abnormally. Control does not return to the next line after `oa.ma(i)` in the `try` block; instead, execution jumps to the matching `catch` in `B.mb`. After that catch block runs, `B.mb` finishes and `Tester.main` resumes after calling `B.mb`. This is the central difference between normal and abnormal execution flow.

Using Eclipse to verify the trace

The lecture also demonstrated how to reproduce the trace in Eclipse. Jackie placed breakpoints at the first executable line of each method, not on method headers, and used the debugger to follow `Tester.main` → `B.mb` → `A.ma`. Jackie also used the expressions panel to check values such as `i` and $i < 0$. The important habit is to predict the next location before clicking in the debugger, then use Eclipse to confirm or correct the paper trace.

Version 2 abnormal flow and compiler enforcement

For Version 2 with a negative input, `A.ma` throws and terminates abnormally, and `B.mb` also terminates abnormally because it specified rather than caught the exception. The exception then reaches `Tester.main`, where the catch block handles it. A student question led Jackie

to emphasize the compile-time rule: if a caller invokes a method that specifies a checked exception but the caller neither catches nor specifies it, the code does not compile.

Bank and Account: resolving which method is called

The Bank/Account example shifted attention from only exception syntax to method-call analysis. The call `b.withdraw(...)` uses a `Bank` object as the context object, so it calls `Bank.withdraw`. Inside `Bank.withdraw`, a call such as `accounts[i].withdraw(...)` uses `accounts[i]` as the context object. Since `accounts` is an `Account[]` and an indexed element is an `Account`, this call resolves to `Account.withdraw`. This distinction matters when tracing both the call stack and exception propagation.

Multiple catch blocks

The lecture then considered a `try` block that may throw different custom exceptions, such as an invalid transaction exception from a withdrawal and a radius-related exception from a circle operation. Under the lecture's simplifying assumption that the custom exceptions are unrelated, their catch blocks may be written in either order. Runtime still checks catch blocks from top to bottom: the first matching catch block executes, and the remaining catch blocks are skipped. Also, once one statement in the `try` block throws, the rest of that `try` block is bypassed; Java does not go back to look for more exceptions in the same try block.

Assigned tracing exercises

Jackie ended by assigning three exercises before the next lecture: watch the short video on normal versus abnormal execution flow for catch-or-specify, trace a circle example where `setRadius` may throw inside a loop, and try the `Integer.parseInt` example that catches `NumberFormatException` when the user enters non-integer input. These exercises are meant to make the exception flow feel mechanical before the course moves to exception assertions in JUnit.

Key Terms, Notation, and Structures

- **Catch-or-specify requirement**: if a method call may throw a checked exception declared by the callee, the caller must either catch it with `try-catch` or declare it with `throws`.
- **Exception loop closure**: the point in the call chain where the exception is caught; callers above that point no longer need to catch or specify that exception.
- **Normal execution flow**: the path where the risky method call does not throw, the `try` block continues, and associated `catch` blocks are skipped.
- **Abnormal execution flow**: the path where a `throw` interrupts the current method, bypasses the rest of the current `try` block, and transfers control to a matching `catch` or to the caller.

- `throw` **vs.** `throws`: `throw` appears inside a method body to signal an exception at runtime; `throws` appears in a method header to specify that the method may pass the exception to its caller.
- **First matching catch**: catch blocks are checked top to bottom; the first catch whose type matches the thrown exception executes, and the remaining catch blocks are ignored.
- **Context object**: the expression to the left of the dot in a method call; it determines which class's method is being invoked, such as `Bank.withdraw` versus `Account.withdraw`.
- **Call stack**: the stack of active method calls; it helps trace entry, suspension, abnormal termination, and resumption across calls such as `Tester.main` → `B.mb` → `A.ma`.
- `NegativeValueException`: the custom checked exception used in the A/B/Tester examples when a negative input value should signal an error.
- `InvalidTransactionException`: the custom exception used in the Bank/Account example to represent an invalid withdrawal or transaction scenario.
- **Radius-related exception**: the custom exception used in the Circle examples when an invalid radius is supplied; use the exact exception name from the code you are tracing.
- `NumberFormatException`: the built-in exception relevant to `Integer.parseInt` when a string such as `"twenty-three"` cannot be converted to an integer.

Key Examples and Demonstrations

Version 1: `B.mb` catches

Setup: `Tester.main` calls `B.mb`, and `B.mb` calls `A.ma`.
`A.ma` declares `throws NegativeValueException`; `B.mb` wraps the call in a `try-catch` block.

Main point: The call `oa.ma(i)` in `B.mb` is subject to catch-or-specify because `A.ma` specifies the exception. The later call to `B.mb` from `Tester.main` is not subject to that exception because `B.mb` already catches it.

What to practise: Trace the same code twice: once with a non-negative `i` and once with a negative `i`. Mark exactly where the normal path resumes and where the abnormal path jumps into the catch block.

Version 2 and Version 3: shifting responsibility

Setup: In Version 2, `B.mb` specifies the exception and `Tester.main` catches it. In Version 3, `B.mb` and `Tester.main` both specify it.

Main point: Specifying does not solve the exception; it moves responsibility to the caller. Catching stops propagation. If every method specifies and no method catches, the exception can reach the console.

What to practise: For each version, draw the call stack and mark where the exception is thrown, where it is propagated, and where it is caught or left uncaught.

Eclipse debugger trace

Setup: Breakpoints are placed at the first executable line of `Tester.main`, `B.mb`, and `A.ma`. The value of `i` is changed between positive and negative inputs.

Main point: The debugger confirms the paper trace: a non-negative input follows the else branch and returns normally; a negative input throws and jumps to a matching catch. Jackie emphasized predicting the next step before using the debugger.

What to practise: Reproduce the demonstration with the posted code. Use the Expressions panel to evaluate `i` and $i < 0$, and compare the debugger's next line with your own paper prediction.

Bank and Account withdrawal calls

Setup: `BankApplication.main` calls `b.withdraw(...)`, and inside the bank method, the code calls `accounts[i].withdraw(...)`.

Main point: The first call invokes `Bank.withdraw` because `b` is a `Bank`. The second call invokes `Account.withdraw` because `accounts[i]` is an `Account`, even though `accounts` itself is an array.

What to practise: Annotate each method call with its caller and callee. Then trace a case where an invalid amount causes an exception and compare it with a case where the withdrawal succeeds.

Multiple catch blocks

Setup: A try block contains multiple operations that may throw different custom exceptions, followed by multiple catch blocks.

Main point: Once one statement in the try block throws, the rest of that try block is skipped. Java then checks catch blocks in order and executes only the first matching one.

What to practise: Use two test cases: one where the withdrawal is invalid but the radius is valid, and one where the withdrawal is valid but the radius is invalid. Label which catch block is reached in each case and which lines are bypassed.

Loop and parsing exercises

Setup: The assigned Circle exercise wraps a radius-setting operation inside a loop, and the parsing exercise uses `Integer.parseInt` inside a loop that catches invalid input.

Main point: Catching an exception inside a loop can allow the loop to continue instead of terminating the whole program. This is a practical pattern for robust user input.

What to practise: Trace the input sequence `5`, `-5`, `10` for the Circle exercise, and trace `"twenty-three"` followed by `"23"` for the parsing exercise.

Common Pitfalls and Misconceptions

Assuming a caller must handle an exception even after the callee catches it

Pitfall: Thinking `Tester.main` must catch or specify `NegativeValueException` in Version 1 even though `B.mb` catches it.

Why it causes problems: It leads to redundant or confused handling and shows that the exception loop has not been understood.

How to avoid it: Check the callee's method header and body: if the callee catches the exception and does not declare `throws`, the caller is not subject to catch-or-specify for that exception.

Continuing the try block after a thrown exception

Pitfall: Tracing the line after `oa.ma(i)` inside the `try` block even when `A.ma` has just thrown an exception.

Why it causes problems: It produces a wrong execution trace: the rest of the try block is bypassed once the exception is thrown.

How to avoid it: When a throw occurs, mark the current method as abnormally terminated and jump to the first relevant catch block or propagate to the caller.

Believing all catch blocks run

Pitfall: After one catch block executes, continuing to the next catch block as if catch blocks were ordinary sequential code.

Why it causes problems: Java executes only the first matching catch block; the remaining catch blocks are skipped.

How to avoid it: In a trace, check catch blocks top to bottom, stop at the first matching one, then continue after the whole try-catch structure.

Ignoring compiler errors from catch-or-specify

Pitfall: Calling a method that declares a checked exception without adding either `try-catch` or a matching `throws` clause.

Why it causes problems: This is a compile-time error, not a runtime issue to be discovered later.

How to avoid it: Treat Eclipse's unhandled-exception message as a design prompt: decide whether this method should catch the exception or propagate it.

Confusing method names across classes

Pitfall: Seeing `withdraw` and assuming all calls go to the same method.

Why it causes problems: In the Bank/Account example, `b.withdraw(...)` and `accounts[i].withdraw(...)` call different methods because their context objects have different types.

How to avoid it: For every method call, identify the full callee as `ClassName.methodName` by checking the type of the expression to the left of the dot.

Letting checked exceptions escape `main` without a reason

Pitfall: Using the Version 3 pattern as if declaring `throws` on `main` were a normal final solution.

Why it causes problems: The exception may be thrown to the console, leaving the user with a raw stack trace and no controlled recovery.

How to avoid it: Catch exceptions somewhere meaningful in the call chain, often where the program can report, recover, retry, or stop gracefully.

Treating the `try` keyword as the main executable step

Pitfall: Expecting the debugger to pause on the `try` keyword itself as the next meaningful step.

Why it causes problems: It creates off-by-one confusion between paper traces and Eclipse traces.

How to avoid it: In paper traces, be clear whether Jackie has numbered the `try` line. In Eclipse, expect the debugger to jump to the first executable statement inside the block.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain when a method call is subject to the catch-or-specify requirement;
- distinguish the roles of `throw` and `throws` in the A/B/Tester examples;
- compare Version 1, Version 2, and Version 3 in terms of where the exception is caught or propagated;
- trace normal execution through `Tester.main`, `B.mb`, and `A.ma` when no exception is thrown;
- trace abnormal execution when `A.ma` throws `NegativeValueException`;
- identify which lines in a `try` block are bypassed after a throw;
- determine which catch block runs when multiple catch blocks are present;
- use the context object to identify whether a call is to `Bank.withdraw`, `Account.withdraw`, `B.mb`, or `A.ma`;
- use Eclipse breakpoints and the debugger to verify a paper trace;

- explain why propagating an exception out of `main` is usually poor design;
- trace a loop that catches an exception and continues to the next iteration;
- use `Integer.parseInt` with `NumberFormatException` as a model for robust input handling;
- plan your immediate preparation around Lab 1, the assigned exception exercises, and the upcoming Programming Test 1.

Suggested Practice

- Recreate the Version 1, Version 2, and Version 3 examples in Eclipse from the posted code, then run each version with both positive and negative input values.
- For Version 1, draw two complete traces: one where `i` is non-negative and one where `i` is negative. Mark normal returns, abnormal termination, catch execution, and caller resumption.
- Delete a required `try-catch` or `throws` clause temporarily and read the Eclipse compiler error. Then fix it using both possible design choices in separate versions.
- In the Bank/Account example, annotate each `withdraw` call with the context object's type and the full callee name.
- Practise the multiple-catch traces using the two lecture test cases: illegal withdrawal with valid radius, and legal withdrawal with invalid radius.
- Complete the assigned normal-versus-abnormal execution-flow video exercise before the next class.
- Trace the Circle loop exercise with the input sequence `5`, `-5`, `10`. Pay attention to whether the loop continues after the exception is caught.
- Try the `Integer.parseInt` exercise using `"twenty-three"` followed by `"23"`; explain why the first input stays in the loop and the second can exit it.
- Continue Lab 1 before looking for solutions, and use the lab's test cases as active preparation for Programming Test 1.

Where to Review

- Review the iPad/PDF notes page 2 for the concrete logistics: Programming Test 1 on Monday, June 8, Lab 1 due Wednesday, June 3, and the reminder that the guide and practice test will be released.
- Review pages 3–5 for Version 1, Version 2, and Version 3 of the A/B/Tester exception-handling chain.

- Review the recording segment where Jackie traces Version 1 on paper and then repeats the normal and abnormal paths in Eclipse.
- Review page 6 for the Bank/Account call-stack example and the distinction between `Bank.withdraw` and `Account.withdraw`.
- Review page 7 for the multiple-catch-block example and the two test cases involving invalid transactions and radius values.
- Review pages 8–13 for the assigned exercises on catch-or-specify execution flows, Circle loops, and parsing strings as integers.
- Revisit the posted exception code on the lecture site, especially the versions used in class, rather than relying only on this digest.

Closing Reminder

This digest is only a roadmap. To learn this lecture properly, you need to replay the traces yourself, compare paper traces with Eclipse debugging, revisit Jackie's iPad notes and the recording, and complete the assigned exercises. Exception handling becomes much easier when you can predict the next executed line before the debugger shows it to you.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 9, June 2

Testing Exceptions in JUnit, Example of Bounded Counters

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- Lecture 9 began the transition from programming exceptions to *testing* exception behaviour with JUnit.
- Jackie first generalized the catch-or-specify rule on a symbolic call stack, identifying which methods are subject to the requirement and which methods are free from it once the exception is caught.
- The running example was a bounded `Counter` whose legal values are from `MIN_VALUE = 0` to `MAX_VALUE = 3`, with `increment()` and `decrement()` able to throw boundary exceptions.
- A state diagram was used to organize test scenarios into normal transitions, where no exception should occur, and boundary cases, where a specific exception should occur.
- The main testing principle was that an exception is not automatically a test failure: the result depends on what the test expects for that scenario.
- Jackie introduced the JUnit structure of a test method, the method under test, expected versus actual values, assertion failures, unexpected exceptions, and the special `fail()` assertion.
- The detailed worked test was for a case where `ValueTooLargeException` is *not* expected after constructing a counter and calling `increment()` once.
- The administrative priority was to finish Lab 1, use the Programming Test 1 practice test under timed conditions, and remember that Lab 2 and exceptions are not part of Programming Test 1.

Roadmap

Announcements and preparation priorities

Jackie opened with several study and deadline reminders. Programming Test 0 grading materials had been sent to the TAs, with marks and feedback expected on Friday. Lab 1 was due Wednesday, June 3, and its solution would be released as a tutorial video after the deadline. Lab 2 would be released soon and would practise exceptions, but Lab 2 and exceptions would not be covered on Programming Test 1. Programming Test 1 is on Monday, June 8, based on Lab 1; Jackie strongly recommended attempting the released practice test with an 80-minute timer before looking for the solution.

Generalizing catch-or-specify on a call stack

Before starting JUnit, Jackie revisited the catch-or-specify requirement using a symbolic call stack with methods `C1.m1`, `C2.m2`, `...`, `Cn.mn`. In the diagram, the exception originates at `C1.m1` near the top of the stack, and `Cn.mn` is the entry point at the bottom. If `Ci.mi` is the

first method that catches the exception, then the methods from `C1.m1` through `Ci.mi` are subject to catch-or-specify, while the methods below `Ci.mi` are free from the requirement because the exception has already been handled.

Only one method catches a propagated exception

A major warning in this review was that only one method in the call chain normally catches a particular propagated exception. Once a method catches it and does not re-throw it, the loop is closed. If a later caller still tries to catch that same exception even though its callee no longer declares it, the result is a compilation error. Jackie suggested experimenting with the earlier Version 1 exception example to see this error concretely.

Introducing the bounded counter

The new JUnit topic used a `Counter` class with static bounds `MIN_VALUE = 0` and `MAX_VALUE = 3`. These static constants represent a global constraint shared by all counter objects. A new counter starts at the minimum value: `increment()` may throw `ValueTooLargeException`, and `decrement()` may throw `ValueTooSmallException`. Because these methods declare checked exceptions, any test calling them must still obey catch-or-specify. A call to `getValue()` returns the current value.

Using a state diagram to find test cases

Jackie drew the legal states of the counter as $0, 1, 2, 3$. Normal `increment()` transitions go $0 \to 1$, $1 \to 2$, and $2 \to 3$. Normal `decrement()` transitions go $3 \to 2$, $2 \to 1$, and $1 \to 0$. The two exceptional boundary cases are `increment()` at state 3, where `ValueTooLargeException` is expected, and `decrement()` at state 0, where `ValueTooSmallException` is expected. The point of the diagram is to turn behaviour into test scenarios systematically rather than guessing tests randomly.

Testing depends on expectation

The lecture then separated test design from implementation behaviour. A test has an expectation: perhaps a particular exception should occur, or perhaps it should not occur. The implementation may or may not actually throw that exception. Whether the test should pass depends on whether the implementation's behaviour matches the expectation. This is why the same event, such as `ValueTooLargeException` being thrown, can mean pass in one test and fail in another.

JUnit test method versus method under test

Jackie next unpacked the structure of a JUnit test. The method marked with `@Test` acts as a caller that coordinates method calls. The method being checked, such as `getValue()` or `increment()`, is the method under test. In a typical assertion such as `assertEquals(23, o.getValue())`, 23 is the expected value and `o.getValue()` produces the actual value. The

JUnit result is a pass or fail depending on whether the expectations encoded by the test are met.

What technically makes a JUnit test fail

Jackie reviewed the technical definition from the slides: a JUnit test fails if an assertion failure occurs, including a deliberate `fail()` call, or if an unexpected exception is thrown. If neither happens, the test passes. This also explains why the default Eclipse/JUnit test stub containing `fail("Not yet implemented")` fails immediately, and why simply deleting or commenting out that line makes the empty test pass while proving nothing useful.

Test 1: `ValueTooLargeException` not expected

The detailed test case focused on a newly constructed counter whose value is 0. Calling `increment()` should move it to 1, so `ValueTooLargeException` is not expected. The test places `c.increment()` inside a `try` block, checks `assertEquals(1, c.getValue())` immediately afterward, and puts `fail("ValueTooLargeException is not expected.")` inside the catch block. The `fail()` is in the catch block because reaching that block means the unexpected exception occurred.

Tracing the test on correct and incorrect implementations

For the correct implementation, `increment()` throws only when the value already equals the maximum. Starting from 0, the condition is false, the value becomes 1, the catch block is skipped, and the test passes. Jackie then changed the implementation to an incorrect condition that throws when the value is less than the maximum. Starting from 0, the exception is thrown unexpectedly, the rest of the `try` block is bypassed, the matching catch runs, `fail()` executes, and the test fails. Jackie summarized the quality criterion: a good test should pass on a correct implementation and fail on an incorrect implementation.

Deferred items

Although the notes list regression testing and a second exception-testing example, Jackie explicitly moved regression testing to Thursday and ended after the first detailed test. The closing advice was to spend a short focused review period on this example before the next lecture, where more exception-testing cases would continue.

Key Terms, Notation, and Structures

- **Catch-or-specify requirement**: for checked exceptions, a caller must either catch a declared exception with `try-catch` or declare it with `throws`.
- **Origin of an exception**: the method where the exception is first thrown, such as `C1.m1` in Jackie's symbolic call-stack diagram.

- **First catcher**: the first method, moving downward through the call stack from the origin toward the entry point, that catches the propagated exception.
- **Methods subject to catch-or-specify**: in the generalized stack, the methods from `C1.m1` through `Ci.mi` when `Ci.mi` is the first catcher.
- **Methods free from catch-or-specify**: the methods from `C(i+1).m(i+1)` through `Cn.mn`, because the exception has already been caught above them.
- `Counter`: the bounded-counter class used to study JUnit tests for normal and exceptional behaviours.
- `MIN_VALUE` **and** `MAX_VALUE`: static bounds on every counter object; in the lecture example they are 0 and 3.
- `increment()`: a method that should increase the counter by one unless the value is already at `MAX_VALUE`, in which case `ValueTooLargeException` is expected.
- `decrement()`: a method that should decrease the counter by one unless the value is already at `MIN_VALUE`, in which case `ValueTooSmallException` is expected.
- **State diagram**: a diagram whose states are possible values of a variable and whose arrows are transitions caused by method calls.
- **JUnit test method**: a method annotated with `@Test`; in this lecture it acts as the caller that coordinates calls to methods under test.
- **Method under test**: the implementation method being checked by a test, such as `getValue()` or `increment()`.
- **Expected value and actual value**: in `assertEquals(expected, actual)`, the first argument is what the test expects, and the second is what the code actually produces.
- **Assertion failure**: a failed assertion such as `assertEquals(...)` or an explicit `fail(...)`; it causes the JUnit test to fail immediately.
- **Unexpected exception**: an exception that the test did not expect; this is another way for a JUnit test to fail.
- `fail()`: a JUnit assertion used when reaching a point in the test already means the expected behaviour has been violated.

Key Examples and Demonstrations

General call-stack diagram for catch-or-specify

Setup: Jackie used a symbolic stack `C1.m1`, `C2.m2`, `...`, `Cn.mn`, with an exception thrown in `C1.m1` and first caught in `Ci.mi`.

Main point: The catch-or-specify requirement applies exactly along the portion of the call chain from the origin to the first catcher; once caught, lower callers are not responsible for that exception.

What to practise: Redraw the diagram, choose a concrete value of n and i, and list the methods that are subject to the requirement versus the methods that are free from it.

Bounded-counter state diagram

Setup: The counter's legal values are $0, 1, 2, 3$. Calls to `increment()` and `decrement()` move between these states or trigger an exception at a boundary.

Main point: A state diagram can reveal both normal tests and boundary tests. Six transitions are normal, while `increment()` from 3 and `decrement()` from 0 are exceptional cases.

What to practise: Recreate the diagram without looking, then write beside each arrow whether the relevant exception is expected or not expected.

JUnit test method versus method under test

Setup: A simple test creates an object and checks the actual return value of a method `getValue` against the expected value: `assertEquals(23, o.getValue())`.

Main point: The test method is not the same as the implementation method. The test method coordinates execution and checks expectations; the method under test produces the actual behaviour being judged.

What to practise: For each JUnit test you write, explicitly identify the caller test method, the method under test, the input arguments, the expected result, and the actual result.

Default failing JUnit test

Setup: Eclipse/JUnit may generate a default test containing `fail("Not yet implemented")`.

Main point: Removing `fail()` makes the test pass only because the test does nothing. This illustrates why the number of passing tests is not meaningful without test quality.

What to practise: Create a small test class, run the default failing test, comment out the `fail()` line, and explain why the green bar is not evidence of correctness.

Test 1 on the correct `increment()` implementation

Setup: A new counter starts at 0.
The test calls `c.increment()` and expects no `ValueTooLargeException`; afterward the value should be 1.

Main point: Since the correct implementation does not throw, execution continues after `c.increment()`, the assertion on value 1 passes, the catch block is skipped, and the test passes.

What to practise: Trace the test line by line, marking when the method under test is entered, when it returns, and why the catch block is not executed.

Test 1 on an incorrect `increment()` implementation

Setup: Jackie changed the implementation so that `increment()` became incorrect: it throws `ValueTooLargeException` when the value is less than the maximum, which is wrong for a newly constructed counter.

Main point: The unexpected exception interrupts the `try` block, jumps to the matching catch block, executes `fail()`, and makes the test fail. This is the desired reaction of a good test to an incorrect implementation.

What to practise: Repeat the trace and explain why the assertion after `c.increment()` is not reached when the exception is thrown.

Common Pitfalls and Misconceptions

Trying to catch an exception after it has already been caught

Pitfall: Adding a `try-catch` in a later caller for an exception already caught by an intermediate method.

Why it causes problems: If the intermediate method no longer declares that exception, the later caller has no such checked exception to catch, and Java reports a compilation error.

How to avoid it: Check the callee's method header. If it does not declare the exception, do not add a catch for that exception at the caller unless the callee can actually throw it.

Treating every exception as an automatic test failure

Pitfall: Thinking that if an exception occurs during a JUnit test, the test must fail.

Why it causes problems: Some tests are written specifically to check that an exception is thrown. In those tests, the expected exception is evidence of correct behaviour.

How to avoid it: Before writing the test body, state the expectation: is this exception expected, or not expected, for this scenario?

Trusting a green bar without checking test quality

Pitfall: Assuming that many passing tests automatically prove the implementation is correct.

Why it causes problems: An empty test passes because nothing can fail, but it checks no behaviour. A large number of weak tests can still miss important errors.

How to avoid it: For each test, identify the scenario, expected behaviour, method under test, and reason the test would fail on a plausible incorrect implementation.

Putting `fail()` in the wrong place

Pitfall: Placing `fail()` without connecting it to a specific violated expectation.

Why it causes problems: `fail()` unconditionally fails the test, so careless placement can make a correct implementation fail or make the test hard to understand.

How to avoid it: In a "no exception expected" test, put `fail()` inside the catch block for the unexpected exception, with a message explaining why reaching that catch is wrong.

Assuming execution continues inside the `try` block after a throw

Pitfall: Expecting the assertion after `c.increment()` to run even when `c.increment()` throws.

Why it causes problems: A throw interrupts the rest of the `try` block and transfers control to the matching catch block.

How to avoid it: Trace two flows separately: the normal flow where the method returns, and the abnormal flow where the method throws and the rest of the `try` block is bypassed.

Confusing boolean negation with assertion commands

Pitfall: Trying to "negate" an assertion such as `assertTrue(...)` as if the assertion itself were a boolean expression.

Why it causes problems: Assertions are commands that pass or fail; the boolean expression is inside the assertion call.

How to avoid it: Use `assertFalse(condition)` when the expected boolean value is false, and reserve logical negation for the condition itself when appropriate.

What You Should Be Able to Do

After this lecture, you should be able to:

- trace a checked exception through a symbolic call stack and identify the origin, first catcher, entry point, and methods subject to catch-or-specify;
- explain why only one method normally catches a particular propagated exception unless the exception is deliberately re-thrown;
- justify why `MIN_VALUE` and `MAX_VALUE` in the counter example are static class-level constraints;
- draw the bounded-counter state diagram and classify each transition as normal or exceptional;
- distinguish a JUnit test method from the method under test, including caller/callee roles, inputs, expected values, and actual values;
- state the two technical causes of a JUnit test failure: assertion failure and unexpected exception;
- explain why an exception can mean pass in one test and fail in another depending on the test expectation;
- write and trace a JUnit test for a "no exception expected" scenario using `try`, `catch`, and `fail()`;
- compare a good implementation and a plausible wrong implementation, and explain how the same test should react to both;

- use the released Programming Test 1 guide and practice test strategically while finishing Lab 1 on time.

Suggested Practice

- Finish Lab 1 before the Wednesday, June 3 deadline, then use the released solution video afterward to compare your approach rather than replacing your own attempt.
- Attempt the Programming Test 1 practice test under an 80-minute timer before looking for the solution; use office hours if you get stuck after a serious timed attempt.
- Redraw the generalized call-stack diagram with concrete values, such as $n = 6$ and $i = 3$, and list which methods are subject to catch-or-specify.
- Revisit the earlier Version 1 exception example and try adding a catch block in a later caller for an exception already caught in the intermediate method; observe and explain the compilation error.
- Redraw the bounded-counter state diagram and turn each arrow or boundary loop into a candidate test scenario.
- Write down, before coding, whether each counter test expects `ValueTooLargeException`, expects `ValueTooSmallException`, or expects no exception.
- Recreate Test 1 for "`ValueTooLargeException` not expected" and trace it on both the correct and incorrect `increment()` implementations from the lecture.
- Create a trivial default JUnit test containing `fail("Not yet implemented")` and then comment it out; explain why the resulting pass is not a useful test.
- Review the self-study slides on creating/running JUnit tests and common assertions, especially if Lab 0 Part 1 or Part 2 felt shaky.

Where to Review

- **iPad/PDF notes, page 2**: deadline and preparation reminders for Programming Test 1, Lab 1, Lab 2, and the practice test.
- **iPad/PDF notes, page 3**: the generalized catch-or-specify call-stack diagram, including origin, first catcher, subject methods, free methods, and extreme cases.
- **iPad/PDF notes, pages 4–6**: the bounded `Counter` class and the quick review of specify versus catch.
- **iPad/PDF notes, page 7**: the state diagram for counter values 0 through 3 and the normal versus exceptional cases.

- **iPad/PDF notes, page 8**: the relationship between a JUnit test method, the method under test, expected values, and actual values.
- **iPad/PDF notes, page 9**: self-study slide ranges for creating/running JUnit tests and common assertions.
- **iPad/PDF notes, page 10**: the definition of JUnit pass/fail and the default failing test generated with `fail("Not yet implemented")`.
- **iPad/PDF notes, page 11**: the expectation matrix comparing test expectations against implementation behaviour.
- **iPad/PDF notes, pages 12–14**: Test 1 for `ValueTooLargeException` not expected, including traces on correct and incorrect implementations.
- **Lecture recording**: revisit the spoken explanations around why `fail()` belongs in the catch block for this test, why a good test must fail a wrong implementation, and why the practice test should be attempted under timed conditions.

Closing Reminder

This digest gives you the roadmap, but the real learning comes from tracing and coding. Revisit Jackie's notes and the lecture recording, finish Lab 1, use the Programming Test 1 guide and practice test carefully, and practise writing JUnit tests whose expectations are clear before you run them. Reading the digest alone is not enough: you should be able to redraw the diagrams, explain the catch-or-specify reasoning, and trace the JUnit test on both correct and incorrect implementations.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 10, June 4

Mixing Expectations in a Test, Console vs. JUnit Tests, Using Loops in a Test

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- Lecture 10 continued the JUnit exception-testing topic by focusing on how test structure changes when exceptions are expected, unexpected, or expected only in a later phase of the same test.
- Jackie contrasted two key patterns: when no exception is expected, put `fail()` in the `catch` block; when an exception is expected, put `fail()` immediately after the method call inside the `try` block.
- The first detailed example tested `decrement()` at the lower bound of a bounded `Counter`, where `ValueTooSmallException` should occur.
- The second detailed example tested `increment()` across two phases:
  first reaching the maximum value without an exception, then calling `increment()` again when `ValueTooLargeException` should occur.
- Correct and incorrect implementations were traced to show how exact boundary conditions such as ==, <, and > affect whether a test passes or fails.
- Jackie emphasized that `System.out.println("Error ...")` is not a substitute for a JUnit `fail()` assertion, because printing does not stop the test execution flow.
- A single `try-catch` block is inappropriate when one phase expects no exception and another phase expects the same exception, because the shared `catch` block cannot tell which phase caused the exception.
- The final example showed how loops can test many normal-path counter values efficiently, while still requiring separate `catch` blocks for each checked exception that may be thrown.
- The main preparation reminders were Programming Test 1 on Monday, June 8, the released guide and practice test, the released Lab 1 solution video, and the newly released Lab 2.

Roadmap

Announcements and assessment planning

Jackie opened with reminders about Programming Test 1, Lab 1, Lab 2, and Programming Test 0. Programming Test 1 is on Monday, June 8 during the scheduled lab session, with the same policy as before and no questions during the invigilated session. The guide and practice test had already been released, and Jackie strongly advised students to attempt the practice test before waiting for the solution. Lab 1's solution video had been released, and Lab 2 had also been released, with about three weeks before it is due. Jackie warned that Lab 2 is more challenging because it involves exceptions, reading JUnit tests, and more advanced array manipulation.

Recap: no exception expected

The lecture began technically by recalling the previous example: a newly constructed `Counter` has value 0, and calling `increment()` should normally move it to 1.

Since `ValueTooLargeException` is not expected, the test puts the method call in a `try` block, verifies the post-state afterward, and places `fail()` in the `catch` block. If the catch block is reached, the implementation has thrown an exception unexpectedly, so the test should fail immediately.

Test 2: exception expected from `decrement()` at the minimum

Jackie then introduced the opposite expectation. A newly constructed counter starts at the minimum value 0, so calling `decrement()` should trigger `ValueTooSmallException`. In this case, the `fail()` assertion belongs immediately after `c.decrement()` inside the `try` block. If execution reaches that line, the expected exception did not occur. If the exception is thrown and the matching `catch` block is reached, the catch block can do nothing, because reaching it means the expected behaviour occurred.

Tracing Test 2 on correct and incorrect `decrement()` implementations

For the correct implementation, `decrement()` checks whether the value is equal to the minimum. Starting from 0, the condition is true, `ValueTooSmallException` is thrown, the `fail()` line is bypassed, the matching `catch` block is reached, and the test passes. Jackie then changed the condition to an incorrect one that would only throw when the value is below the minimum. Starting from 0, no exception is thrown, the counter incorrectly moves to -1, execution reaches `fail()`, and the test fails immediately.

Test 3: one test with two different exception expectations

The next example used `increment()` in two phases. Phase 1 calls `increment()` three times to move the counter from 0 to 3; during this phase, `ValueTooLargeException` is not expected. Phase 2 calls `increment()` once more when the counter is already at its maximum; during this phase, `ValueTooLargeException` is expected. Jackie separated these phases into two `try-catch` blocks because the meaning of the same exception changes between the phases.

Tracing Test 3 on a correct implementation

For the correct `increment()` implementation, the method throws only when the current value already equals `Counter.MAX_VALUE`. The first three calls move $0 \to 1$, $1 \to 2$, and $2 \to 3$ without throwing. The test then checks that the value is the maximum. The fourth call is made at value 3, so the expected `ValueTooLargeException` is thrown, the `fail()` after that call is bypassed, the second catch block is reached, and the test passes.

Tracing Test 3 on two incorrect boundary checks

Jackie then used two incorrect versions of `increment()` to show how the same test can fail for different reasons. If the implementation throws when `value < Counter.MAX_VALUE`, the exception occurs too early during Phase 1, where it is not expected, so the first catch block calls `fail()`. If the implementation throws only when `value > Counter.MAX_VALUE`, Phase 1 appears fine, but the fourth call at value 3 incorrectly increments to 4 instead of throwing, so the `fail()` in Phase 2 is reached.

Why console testers do not replace JUnit tests

Jackie next compared a console-style tester with the JUnit version. In the console tester, errors are signalled by `System.out.println("Error ...")`, not by `fail()`. The problem is that printing an error message does not stop execution. Jackie traced a wrong `increment()` implementation where an early unexpected exception leads to an error message, but the tester continues and can later print a success message too. This makes the output contradictory and much less reliable than a JUnit test that fails immediately.

Why not combine mixed expectations into one `try-catch`

The next exercise asked whether the two phases of Test 3 could be merged into one `try-catch` block. Jackie explained that this is ambiguous. If `ValueTooLargeException` is thrown during Phase 1, the test should fail; if the same exception is thrown during Phase 2, the test should pass. A single shared `catch` block cannot tell which phase produced the exception unless the test is redesigned. The correct habit is to isolate phases with different expectations into separate `try-catch` blocks.

Testing many normal values with loops

The final example showed how a loop can avoid writing many repeated calls when the range of legal counter values is large. The test loops through normal `increment()` transitions and then normal `decrement()` transitions, checking the value after each operation. Because `increment()` may throw `ValueTooLargeException` and `decrement()` may throw `ValueTooSmallException`, the test needs catch blocks for both checked exceptions. If one catch block is removed, Eclipse reports a compilation error because the corresponding checked exception is not handled.

Key Terms, Notation, and Structures

- **Expected exception**: an exception that the test is intentionally trying to observe; if it occurs in the intended place, the test may pass.
- **Unexpected exception**: an exception that should not occur for the scenario being tested; if it occurs, the test should fail.

- `fail()`: a JUnit assertion that fails the test immediately and unconditionally when execution reaches it.
- `ValueTooSmallException`: the custom checked exception expected when `decrement()` is called while the counter is already at its minimum value.
- `ValueTooLargeException`: the custom checked exception expected when `increment()` is called while the counter is already at its maximum value.
- `try-catch` **placement**: the location of `fail()` depends on the expectation: in the `catch` block when the exception is not expected, and immediately after the call when the exception is expected.
- **Phase**: a segment of a test method with its own expectation; in Test 3, Phase 1 expects no `ValueTooLargeException`, while Phase 2 expects it.
- **Normal execution flow**: execution continues to the next statement after a method call because no exception was thrown.
- **Abnormal execution flow**: execution is disrupted by a thrown exception, so the rest of the `try` block is bypassed and a matching `catch` block is considered.
- **Boundary condition**: the condition at the edge of an inequality (or a Boolean condition in general), such as `value == Counter.MIN_VALUE` or `value == Counter.MAX_VALUE`; using < or > in the wrong place changes the test outcome.
- **Console tester**: a test-like `main` method that prints messages; Jackie used it to show why printing is weaker than JUnit assertions.
- **Loop-based test**: a JUnit test that uses `for` loops to check many similar normal transitions instead of writing repeated method calls manually.
- **Multiple checked exceptions**: if a `try` block calls methods that can throw different checked exceptions, each type must be caught or specified.

Key Examples and Demonstrations

Testing an expected `ValueTooSmallException`

Setup: A newly constructed `Counter` has value 0, and the test calls `c.decrement()`.

Main point: Since `ValueTooSmallException` is expected, the test puts `fail()` immediately after `c.decrement()` in the `try` block. If the exception is thrown, that `fail()` is bypassed and the empty matching catch block lets the test pass.

What to practise: Trace the test twice: once when `decrement()` correctly throws at value 0, and once when it incorrectly allows the value to become -1.

A two-phase test for `increment()`

Setup: The test first calls `c.increment()` three times to move from 0 to 3, then calls `c.increment()` once more at the maximum.

Main point: The same exception has different meanings in the two phases.

In Phase 1, `ValueTooLargeException` is unexpected and should fail the test.

In Phase 2, `ValueTooLargeException` is expected and should make the test pass if it occurs at the correct time.

What to practise: Redraw the two phases and mark exactly where each `fail()` belongs. Then explain why the test needs two separate `try-catch` blocks.

Boundary-condition bugs in `increment()`

Setup: Jackie compared the correct check `value == Counter.MAX_VALUE` with incorrect checks such as throwing too early or too late.

Main point: A wrong comparison can either throw `ValueTooLargeException` before the maximum is reached, or fail to throw it at the maximum. The two-phase test detects both kinds of error, but for different reasons.

What to practise: Change the boundary condition in a small counter implementation and predict whether the failure happens in Phase 1 or Phase 2 before running the test.

Console tester versus JUnit test

Setup: A console-style tester prints messages such as `"Error ..."` and `"Success ..."` instead of using JUnit assertions.

Main point: `System.out.println` does not change the pass/fail status of a test and does not stop execution. A tester can therefore print an error and later print success, which is confusing. `fail()` is the correct JUnit mechanism for an immediate, unambiguous failure.

What to practise: Trace the console tester with an incorrect `increment()` implementation and identify how both an error message and a success message can appear.

Combining catch blocks incorrectly

Setup: Jackie asked whether the two phases of the `increment()` test could be merged into one larger `try` block with one `catch` block.

Main point: A single catch block for `ValueTooLargeException` cannot know whether the exception came from the no-exception-expected phase or the exception-expected phase. Leaving the catch empty is correct for Phase 2 but wrong for Phase 1; putting `fail()` there is correct for Phase 1 but wrong for Phase 2.

What to practise: Write a short explanation of why this design is ambiguous without relying on Eclipse to tell you.

Loop-based normal-path testing

Setup: The final example used loops to test many normal `increment()` and `decrement()` transitions.

Main point: Loops are useful when the legal range is large, such as 0 through 1000, because writing every method call manually is not practical. However, the test still must handle every checked exception that the methods may throw.

What to practise: Convert the small counter test into a loop version, then remove one catch block and explain the resulting compilation error.

Common Pitfalls and Misconceptions

Putting `fail()` in the wrong place

Pitfall: Using the same `fail()` placement for every exception test.

Why it causes problems: If the exception is expected, `fail()` belongs after the method call in the `try` block; if the exception is not expected, `fail()` belongs in the `catch` block. Reversing these patterns makes the test pass and fail for the wrong reasons.

How to avoid it: Before coding, write one sentence: "In this scenario, exception X is expected" or "exception X is not expected." Then place `fail()` based on that sentence.

Treating every exception occurrence the same way

Pitfall: Assuming that a thrown exception always means the test failed, or always means the test passed.

Why it causes problems: The lecture examples show both possibilities.

A thrown `ValueTooSmallException` is correct when decrementing from 0, but a thrown `ValueTooLargeException` is wrong when incrementing from 0 to 1.

How to avoid it: Judge the thrown exception against the test expectation, not against a general rule such as "exceptions are bad."

Using `System.out.println` as a test oracle

Pitfall: Printing `"Error"` or `"Success"` instead of using assertions.

Why it causes problems: Printing does not fail the test and does not stop execution. A flawed console tester can continue after an error and print a later success message.

How to avoid it: Use JUnit assertions such as `assertEquals(...)` and `fail(...)` to encode pass/fail behaviour.

Merging phases with different expectations

Pitfall: Putting a no-exception-expected phase and an exception-expected phase into one shared `try-catch` block.

Why it causes problems: The shared catch block receives the same exception type from different possible locations, but the correct response depends on where it came from.

How to avoid it: Split the test into separate phases with separate `try-catch` blocks whenever the expectation changes.

Using the wrong boundary comparison

Pitfall: Writing a boundary check such as `value < Counter.MAX_VALUE` or `value > Counter.MAX_VALUE` when the intended condition is equality at the boundary.

Why it causes problems: Throwing too early fails normal cases, while throwing too late allows illegal states such as going from 3 to 4.

How to avoid it: Match the condition to the state diagram: throw on `increment()` when already at the maximum, and throw on `decrement()` when already at the minimum.

Forgetting a checked exception in a loop-based test

Pitfall: Calling both `increment()` and `decrement()` in one `try` block but catching only one of their checked exceptions.

Why it causes problems: The Java compiler enforces catch-or-specify. If `decrement()` may throw `ValueTooSmallException` and no matching catch or `throws` clause is present, the code will not compile.

How to avoid it: For each method call inside the `try` block, look at the method header and list every checked exception that must be caught or specified.

What You Should Be Able to Do

After this lecture, you should be able to:

- distinguish a test where an exception is expected from a test where an exception is not expected;
- decide where `fail()` belongs in each of those two cases and justify the placement using execution flow;
- trace `decrement()` at the lower bound on both correct and incorrect implementations;
- trace a two-phase `increment()` test where `ValueTooLargeException` is first not expected and later expected;
- explain how incorrect boundary checks can fail different phases of the same test;
- explain why `System.out.println` is not a reliable substitute for JUnit assertions;
- identify why combining mixed expectations into a single `try-catch` block is ambiguous;
- write loop-based JUnit tests for a range of normal counter values;
- apply catch-or-specify when a test calls methods that declare different checked exceptions;
- prepare for Programming Test 1 using the released guide and practice test, while also starting Lab 2 early.

Suggested Practice

- Attempt the Programming Test 1 practice test under timed conditions before using the released solution.
- Review the Lab 1 solution video after making your own serious attempt, and compare Jackie's development path with your code.
- Start Lab 2 early; pay special attention to the JUnit tests and exception behaviour rather than waiting until the due date approaches.
- Write the two basic exception tests yourself: one where an exception is not expected and one where an exception is expected. Place `fail()` correctly in each.
- Recreate the `decrement()` lower-bound test and trace it on both a correct implementation (e.g., `this.value == Counter.MIN_VALUE`) and an incorrect implementation.
- Recreate the two-phase `increment()` test and trace it on the correct implementation, a throw-too-early implementation, and a throw-too-late implementation.
- Convert a repeated sequence of `increment()` and `decrement()` calls into a loop-based normal-path test.
- Remove one required catch block from the loop-based test and read the compiler error carefully; explain which method call caused the missing exception obligation.
- Explain in writing why the console tester can produce contradictory output and why the combined-catch design is ambiguous.

Where to Review

- **iPad/PDF notes, page 2**: Programming Test 1 reminders, the released guide and practice test, Lab 1 solution video, and Lab 2 release.
- **iPad/PDF notes, page 3**: the JUnit test where `ValueTooSmallException` is expected from `decrement()` at value 0.
- **iPad/PDF notes, pages 4–5**: traces of Test 2 on correct and incorrect `decrement()` implementations.
- **iPad/PDF notes, page 6**: the two-phase `increment()` test where `ValueTooLargeException` is sometimes expected and sometimes not.
- **iPad/PDF notes, pages 7–9**: traces of Test 3 on the correct implementation and two incorrect boundary-condition implementations.
- **iPad/PDF notes, page 10**: the console tester exercise and why printing messages is not equivalent to JUnit failure.

- **iPad/PDF notes, page 11**: the combined-catch exercise and the ambiguity caused by mixing expectations in one `try-catch` block.
- **iPad/PDF notes, page 12**: the loop-based test for many normal values and the need for catch blocks for both checked exceptions.
- **Lecture recording**: revisit Jackie's spoken explanations around `fail()` placement, why a failed assertion stops execution, the console-tester contradiction, and the missing-catch compilation error.
- **Course materials**: use the Programming Test 1 guide and practice test for immediate preparation, and use Lab 2 to practise reading exception-heavy JUnit tests.

Closing Reminder

This digest is only a roadmap. To learn this material, you need to trace the JUnit tests by hand, run small variants in Eclipse, and explain why each test passes or fails under each implementation. Pay special attention to expectation, `fail()` placement, boundary conditions, and catch-or-specify obligations. Use Jackie's notes, the recording, the Lab 1 solution video, the Programming Test 1 practice materials, and Lab 2 together; reading this digest alone is not enough.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 11, June 9

TDD via Regression Testing, Call by Value, and Equality via == vs. `equals`

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

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 11 connected software-engineering testing habits with a new object-equality topic: first by reviewing TDD and regression testing, then by building the call-by-value model needed for equality.
- Jackie emphasized incremental development: start testing as soon as a small logical unit, such as a constructor or method, becomes testable rather than waiting until the whole lab is complete.
- A passing JUnit suite only shows correctness with respect to the tests that were run; better confidence requires more distinct and meaningful tests.
- Regression testing means re-running existing tests after each change to make sure earlier behaviour has not been broken.
- Java method calls are always call by value: before the callee's first executable line, an implicit assignment copies the argument value into the parameter.
- For primitive arguments, the copied value is separate, so changing the parameter does not change the caller's original variable.
- For reference arguments, the copied value is an address, so the parameter becomes an alias of the same object; reassigning the parameter is local, but mutating the shared object is visible through all aliases.
- The lecture previewed object equality by separating ==, which compares primitive values or object references, from `equals`, whose object behaviour depends on whether the class defines a custom version.
- Jackie linked these ideas to upcoming work: Lab 2 remains the immediate priority, object equality will matter for Lab 3 and a future programming test, and Programming Test 2 is listed for July 6.

Roadmap

Announcements and near-term priorities

Jackie began by placing the class in the semester schedule. There are two lectures before the reading week, and the reading week starts on Tuesday. Lab 2 is the main current priority and is due in the week of June 22. The TA will be available in the scheduled lab on the Monday before the reading week for Lab 2 questions, and Jackie expected to be available in person that Monday as well. Programming Test 1 grading was about to begin, with results expected by the end of reading week or shortly after, while Written Test 1 and Programming Test 2 are upcoming. The notes also list Programming Test 2 for July 6.

TDD and incremental development

The first technical segment used the posted TDD workflow diagram. Jackie explained that development and testing should be iterative: develop a small part of the model classes, derive or write test cases, run the tests, fix failures, and repeat. The important timing point is to introduce tests as soon as the code becomes testable. A testable unit may be one constructor, one accessor, or one method; waiting for a whole class or entire lab to be finished makes debugging harder.

Interpreting green and red bars

Jackie then discussed what test results mean. If all tests pass, the code is not proven absolutely correct; it is only correct with respect to the current test suite. Therefore, the number and distinctness of the tests matter. If at least one test fails, it indicates a mismatch between expected and actual outcomes. Jackie again recommended using breakpoints and the debugger rather than relying on print statements.

Regression testing

The lecture then introduced regression testing as a positive software-engineering habit. In ordinary reading, repeatedly going backward may be inefficient, but in software testing, going back and re-running all existing tests after a change is exactly what you want. The purpose is to make sure that a fix or new feature has not broken earlier behaviour. This habit is especially useful for labs, where multiple classes and methods are built over time.

Why object equality starts with call by value

The next topic shifted to object equality. Jackie previewed that equality will matter for Lab 3 and future testing tasks, including implementing a custom `equals` method. Before defining equality carefully, students need a precise method-call model. That is why the lecture first studied call by value, which will also be important later for inheritance.

Caller, callee, parameter, and argument

Jackie reviewed the terminology using a generic example: in `co.m(arg)`, the method `A.m` is the callee if `co` has type `A`, while the enclosing method `B.n` is the caller. The parameter is declared in the callee's method header, while the argument appears at the call site. This distinction is necessary because call by value connects each argument to its corresponding parameter.

The call-by-value rule

The central rule was stated as an implicit assignment: before the callee's first executable line runs, Java performs `parameter = argument`. This happens automatically for every method call. The difference between primitive and reference examples is not whether call by value happens; it always happens. The difference is what value is being copied.

Primitive argument: copying an integer value

The first concrete example used `Circle.setRadius(int r)`. The caller creates a `Circle c`, stores `10` in an `int arg`, and calls `c.setRadius(arg)`. The implicit assignment copies the integer value, so `r` receives its own copy of `10`. The method then assigns `this.radius = r`. This is the simplest case: the parameter receives a primitive value, and no aliasing is involved.

Reference argument: copying an address

The second `Circle` example changed the parameter type to `Circle`. The caller creates one circle `c` with default radius and another circle `arg` with radius `10`, then calls `c.setRadius(arg)`. The implicit assignment `c = arg` in the callee's scope copies an address, not the whole object. The parameter becomes an alias of the argument object, and `this.radius = c.radius` copies the radius value from that aliased object into the context object.

Reassigning a primitive parameter

Jackie next used a `Util` class and a JUnit-style test. The caller has `int i = 10` and calls `u.reassignInt(i)`. Inside the method, the parameter `j` is changed to `j + 1`. Because `j` is a separate copy of `i`, the original `i` remains `10`. Jackie used this to reinforce the idea that modifying a primitive parameter does not modify the caller's variable.

Reassigning a reference parameter

The next example used a `Point p = new Point(3,4)` and a method parameter `Point q`. Call by value first makes `q` an alias of `p`. Inside the method, Jackie then traced an assignment where `q` is reassigned to a newly created `Point(6,8)`. This changes only the address stored in the local parameter `q`; the caller's variable `p` still points to the original `Point(3,4)`. The test's reference and coordinate assertions remain consistent with the original object.

Calling mutators through a reference parameter

The final call-by-value example kept `q` pointing to the same object as `p` and called mutators such as `q.moveHorizontally(3)` and `q.moveVertically(4)`. This time, the caller-visible object changes because `q` and `p` are aliases to the same object. The object state changes from $(3, 4)$ to $(6, 8)$, and those changes are visible through `p` and through any other alias.

Java does not use true call by reference

A student asked whether this reference example should be called call by reference. Jackie clarified that Java still uses call by value. For reference types, the value being copied is the reference value, which is an address. This is different from true call by reference in languages such as C, where an address operator such as `&` can allow a method-like routine to work with the caller's variable itself.

Preview: == versus equals

The lecture ended by introducing two kinds of equality. For primitive values, == compares values, and calling equals on a primitive such as i.equals(j) is a compilation error because i is not an object. For reference values, p1 == p2 compares addresses. If p1 and p2 are two distinct Point(3,4) objects, p1 == p2 is false. The result of p1.equals(p2) depends on the Point class: a custom equals may compare coordinates, while the default inherited version compares references.

Key Terms, Notation, and Structures

- **TDD**: test-driven development; in this lecture, the practical habit of developing code and tests together in an iterative cycle.
- **Incremental development**: building and testing small units as soon as they are testable, rather than waiting until the whole system is complete.
- **Regression testing**: re-running existing tests after making changes, to check that earlier behaviour still works.
- **Green bar**: a JUnit result in which all tests pass; it means the code passed the current tests, not that it is absolutely correct.
- **Red bar**: a JUnit result with at least one failed test; Jackie linked this to a mismatch between expected and actual outcomes and to debugger use.
- **Caller**: the method context that makes a method call, such as B.n when co.m(arg) appears inside n.
- **Callee**: the method being called, such as A.m when the context object co has type A.
- **Parameter**: the variable declared in a method header, such as param in void m(T param).
- **Argument**: the value or expression supplied at the call site, such as arg in co.m(arg).
- **Call by value**: Java's method-call mechanism; each parameter receives a copy of the corresponding argument value.
- **Implicit assignment**: the conceptual step parameter = argument, performed before the callee's first executable line.
- **Primitive value**: a direct value such as an int; copying it gives the callee a separate value from the caller's variable.
- **Reference value**: an address-like value stored in a reference variable; copying it can create an alias to the same object.
- **Alias**: another reference pointing to the same object; changes to the object's state through one alias are visible through the others.

- **Reassigning a parameter**: assigning a parameter variable to point somewhere else; for a reference parameter, this changes only the local parameter, not the caller's variable.
- **Mutator method**: a method that changes an object's state, such as `moveHorizontally`, `moveVertically`, or a setter.
- `==`: for primitives, compares primitive values; for references, compares whether two references point to exactly the same object.
- `equals`: a method call for object comparison; its meaning depends on whether the class defines a custom version or inherits the default version from `Object`.
- **Default** `equals`: the inherited behaviour used when a class does not explicitly define `equals`; Jackie previewed that it behaves like reference comparison.

Key Examples and Demonstrations

TDD workflow and regression testing

Setup: Jackie used the posted workflow diagram with Java model classes, JUnit test cases, test execution, fixing failures, and adding more tests.

Main point: Testing is not a one-time final step. A good workflow alternates between development, testing, debugging, and re-running tests. Passing tests increase confidence only relative to the test suite, while failing tests point to a mismatch that should be investigated with breakpoints and the debugger.

What to practise: For Lab 2, choose one completed method or constructor and write or run tests for it before moving on. After changing the code, re-run earlier tests and check that previous behaviour still passes.

Primitive call by value: `Circle.setRadius(int r)`

Setup: A caller creates `Circle c`, sets `int arg = 10`, and calls `c.setRadius(arg);` the callee has parameter `int r` and assigns `this.radius = r`.

Main point: The implicit assignment copies the integer value: `r = arg`. The method uses the copied value to update the context object's radius. The example establishes the basic call-by-value trace before reference types are introduced.

What to practise: Redraw the call and explicitly write the hidden assignment before tracing the first line of the callee.

Reference call by value: `Circle.setRadius(Circle c)`

Setup: A caller creates one default `Circle c`, another `Circle arg` with radius `10`, and calls `c.setRadius(arg);` the parameter is also named `c` inside the callee.

Main point: The implicit assignment copies the address stored in `arg`, so the parameter becomes an alias of the argument object. The method then reads `c.radius` through that alias and copies the radius into the context object.

What to practise: Draw the two circle objects and be careful to distinguish the caller's context object `c` from the callee's parameter `c`; same name does not mean same variable.

Primitive parameter reassignment in `Util.reassignInt`

Setup: The test creates `int i = 10`, calls `u.reassignInt(i)`, and checks that `i` is still `10`; inside the method, parameter `j` is incremented.

Main point: The method changes `j`, not `i`. Since `j` is a separate primitive copy, the caller's original variable is untouched.

What to practise: Trace the local variable boxes for `i` and `j`. Explain why `j` can become `11` while `i` remains `10`.

Reference parameter reassignment in `Util.reassignRef`

Setup: The caller has `Point p = new Point(3,4)` and a saved alias `refOfPBefore = p`. The method receives `Point q`, creates a new `Point(6,8)`, and assigns `q` to that new point.

Main point: The reassignment changes only the local parameter `q`. It does not change `p`, so the caller still sees the original point with coordinates $(3, 4)$.

What to practise: Redraw the before-and-after reference diagram. Mark which arrow changes and which arrows do not change.

Mutating through a reference parameter in `Util.changeViaRef`

Setup: The caller again passes `p = new Point(3,4)` to a method with parameter `q`, but this time the method calls mutators on `q` instead of reassigning `q`.

Main point: Because `q` and `p` refer to the same object, mutating the object through `q` changes the object visible through `p`. After horizontal and vertical moves, the caller observes coordinates $(6, 8)$.

What to practise: Explain the difference between changing an arrow and changing the object at the end of the arrow.

Primitive and reference equality preview

Setup: For primitives, Jackie compared `int i = 23` and `int j = 46`. For references, Jackie considered two separately constructed `Point(3,4)` objects.

Main point: For primitives, `==` compares values and `i.equals(j)` does not compile. For references, `p1 == p2` compares addresses, while `p1.equals(p2)` depends on the class definition. If `Point` defines a custom `equals`, it may compare coordinates; if it does not, the default inherited behaviour compares references.

What to practise: Create two separate objects with the same attribute values and predict `==` and `equals` under both cases: custom `equals` present and custom `equals` absent.

Common Pitfalls and Misconceptions

Waiting too long before testing

Pitfall: Writing most or all of a lab before running meaningful tests.

Why it causes problems: When many methods and classes are unfinished or untested, a failure is harder to locate and debug.

How to avoid it: Test as soon as a small logical unit is complete. Re-run earlier tests whenever you change a model class.

Treating a green bar as proof of complete correctness

Pitfall: Assuming that all passing tests mean the program is completely correct.

Why it causes problems: Tests only check the scenarios they cover. Repeating the same scenario many times gives less confidence than testing distinct normal and abnormal cases.

How to avoid it: Ask what scenarios are covered, what boundary cases are missing, and whether the tests are meaningfully distinct.

Forgetting the implicit call-by-value assignment

Pitfall: Starting a trace directly inside the callee without first accounting for `parameter = argument`.

Why it causes problems: You may misunderstand which variable is being changed and whether an alias has been created.

How to avoid it: For every method call, write the implicit assignment before tracing the callee body.

Expecting primitive parameter changes to affect the caller

Pitfall: Thinking that changing `j` inside `reassignInt(int j)` changes the caller's `i`.

Why it causes problems: The callee works on a separate copy of the primitive value, so the caller's variable remains unchanged.

How to avoid it: Draw separate boxes for primitive arguments and parameters.

Confusing reference parameter reassignment with object mutation

Pitfall: Thinking that assigning `q = new Point(...)` inside a method changes the caller's variable `p`.

Why it causes problems: The assignment changes only the local parameter's address; the caller's reference still points where it originally pointed.

How to avoid it: Distinguish between changing a reference variable's arrow and changing the object that the arrow points to.

Calling Java reference passing "call by reference"

Pitfall: Saying Java uses call by reference whenever an object reference is passed.

Why it causes problems: This hides the key fact that the reference value itself is copied. Reassigning the parameter cannot reassign the caller's variable.

How to avoid it: Say: Java is call by value; for objects, the copied value is a reference value.

Assuming `equals` always compares object content

Pitfall: Believing `p1.equals(p2)` must be true whenever `p1` and `p2` have matching attribute values.

Why it causes problems: If the class does not define a custom `equals`, the default inherited version compares references, so two distinct objects may be considered unequal.

How to avoid it: Check whether the class explicitly defines `equals`; when writing your own class, implement custom equality if content equality is required.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain the basic TDD cycle and why Jackie recommends incremental development for labs;
- explain what a green bar and red bar do, and do not, tell you about program correctness;
- define regression testing and justify why existing tests should be re-run after code changes;
- distinguish caller from callee and parameter from argument in a method-call example;
- trace a Java method call by explicitly applying the implicit assignment `parameter = argument;`
- distinguish call-by-value behaviour for primitive arguments and reference arguments;
- draw a memory diagram showing aliases created by copying a reference value;
- explain why reassigning a reference parameter does not reassign the caller's variable;
- explain why calling a mutator through a reference parameter can change the caller-visible object state;
- compare Java's call by value with the idea of true call by reference, without calling Java's reference passing call by reference;
- predict the result of == for primitive values and for object references;
- explain why calling `equals` on a primitive causes a compilation error;
- predict how `p1.equals(p2)` depends on whether the class defines a custom `equals` method or inherits the default version.

Suggested Practice

- For Lab 2, choose one method or constructor you have completed and run relevant tests before continuing; then re-run those tests after your next change.
- Redraw the TDD workflow diagram and annotate what you should do when all tests pass versus when at least one test fails.
- For each call-by-value example, write the hidden assignment `parameter = argument` before tracing the method body.
- Recreate the `Util.reassignInt` example and explain why the caller's `i` stays `10`.
- Recreate the two `Point` examples: one where a reference parameter is reassigned and one where mutators are called through the parameter.
- For the `Point` examples, draw variables as boxes and objects as separate structures; mark aliases clearly.
- Create two distinct `Point(3,4)` objects and predict `p1 == p2`; then reason separately about `p1.equals(p2)` when custom `equals` is present versus absent.
- Practise explaining in words: "Java is call by value; when the argument is a reference, the copied value is an address."
- Keep Lab 2 as the immediate priority, and start connecting today's call-by-value diagrams to the upcoming equality work for Lab 3.

Where to Review

- Review the announcement page for current priorities: Lab 2, office-hour timing, and Programming Test 2 on July 6.
- Review the TDD/regression-testing page in the iPad notes, especially the loop from Java classes to JUnit tests, test results, fixes, and adding more tests.
- Review the caller/callee page and practise identifying `A.m` as callee, `B.n` as caller, the parameter in the method header, and the argument at the call site.
- Review the primitive `Circle.setRadius(int)` example to see the simplest form of call by value.
- Review the reference `Circle.setRadius(Circle)` example to see how copying a reference value creates an alias.
- Review the `Util.reassignInt` page for primitive parameter reassignment.
- Review the `Util.reassignRef` page for reference parameter reassignment, especially the note that only the parameter's stored address changes.

- Review the `Util.changeViaRef` page for the mutator-on-alias case, where changes are visible to other aliases.
- Review the equality page introducing == versus `equals`, and prepare to continue from the default inherited `equals` behaviour next class.
- Use the lecture recording for the spoken explanations around aliasing, the student question about call by reference, and Jackie's warnings about custom `equals` for Lab 3 and future tests.

Closing Reminder

This digest is a roadmap for review, not a substitute for the lecture recording, iPad notes, slides, labs, or your own tracing work. The most important active work from this lecture is to draw the call-by-value diagrams yourself: first for primitive values, then for reference aliases, then for reassignment versus mutation. Keep Lab 2 moving, and use the TDD and regression-testing habits as part of how you work, not only as definitions to memorize.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 12, June 11

Default vs. Overridden `equals` (Phases 1, 2), and Short-Circuit Evaluation

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 10

Closing Reminder 10

Main Ideas

- Lecture 12 continued object equality by explaining where `equals` comes from, when the inherited default version is used, and when an overridden version is invoked.
- Every Java class inherits methods from `Object`; if a class such as `PointV1` does not explicitly declare `equals`, it still has the default `Object.equals` method.
- The default `equals` method performs reference comparison: it effectively returns the result of `this == obj`.
- If a class such as `PointV2` overrides `equals`, the inherited default version is no longer the applicable version for objects whose dynamic type is `PointV2`.
- For a call `x.equals(y)`, the version of `equals` invoked is determined by the dynamic type of the context object `x`, not by the argument `y`.
- Jackie used call by value again to explain an overridden `equals(Object obj)` method: `this` refers to the context object, while `obj` receives a copy of the argument reference.
- The first two standard phases of an overridden `equals` method are quick checks: `this == obj` returns `true`, while `obj == null` returns `false`.
- Jackie emphasized that checking `this == null` inside an instance method is not useful because a null context object would cause a `NullPointerException` before the method body is entered.
- The lecture ended with short-circuit evaluation for `&&` and `||`, showing how left-to-right evaluation can be used to guard risky expressions such as division and array access.
- Jackie explicitly connected dynamic `equals` tracing and short-circuit reasoning to tests, exams, and Written Test 1 practice.

Roadmap

Starting from `PointV1`: why `equals` compiles

The lecture opened with a simple `PointV1` class containing only fields and a constructor. Even though `PointV1` does not explicitly declare an `equals` method, a call such as `p1.equals(p2)` still compiles. Jackie used this to introduce the role of inheritance: every class is connected to the `Object` class, and `Object` provides a default `equals` method.

The default method inherited from `Object`

Jackie then showed the default `Object.equals(Object obj)` implementation. Its body is essentially `return this == obj;`, so it compares references rather than object contents. The Eclipse "Step Into" demonstration showed that if `PointV1` does not override `equals`, debugging `p1.equals(p2)` enters `Object.java`. Jackie noted that students who try this themselves may need to attach the JDK source in Eclipse.

Override or not: `PointV1` versus `PointV2`

The next visual structure compared `PointV1`, which does not declare `equals`, with `PointV2`, which declares its own `equals(Object obj)`. The key inheritance idea was: inherited methods are available to child classes, but if the child class redeclares or overrides a method, the overridden child version is used for objects of that dynamic type. Jackie described `Object` as the parent or superclass and the point classes as child or subclass examples, while postponing the full inheritance topic until after the reading week.

Dynamic type decides the method version

Jackie then stated the main dispatch rule for `x.equals(y)`. To decide which `equals` version is invoked, trace the object currently pointed to by `x`. The class name in that object box is the dynamic type of `x`. If that dynamic type overrides `equals`, invoke the overridden version; otherwise, invoke the default version inherited from `Object`. The argument `y` matters for the comparison itself, but not for deciding which method body is selected.

Tracing default `equals` with `PointV1`

Using the `PointV1` examples, Jackie showed that `p1 == p2` is `false` for two separately constructed point objects even when their coordinates match. Since `PointV1` uses the inherited default `equals`, `p1.equals(p2)` also behaves like a reference comparison. A highlighted contrast was that `p1 == s`, where `s` is a `String`, is a compilation error because the reference types are incompatible, while `p1.equals(s)` compiles because `equals` accepts an `Object` argument.

Call by value inside overridden `equals`

The lecture then moved to `PointV2`, where an overridden `equals(Object obj)` is available. Jackie connected this to the previous lecture's call-by-value model: in `p2.equals(p3)`, the argument reference `p3` is copied into the parameter `obj`. Inside the method, `this` refers to the context object `p2`, and `obj` refers to the argument object copied by call by value. This setup is essential before tracing the phases of the method.

Phase 1: same object means equal

The first phase of the overridden method checks `if (this == obj) return true;`. This handles the case where the context object and argument are the same object or aliases of the same object. Jackie emphasized that anything should be equal to itself, so this phase can return `true` without checking fields.

Phase 2: null argument means not equal

The second phase checks `if (obj == null) return false;`. Reaching this phase already means the phase-1 reference-equality test failed. Jackie compared this standard version with a more verbose alternative that checks whether `this` is null, then explained why that extra

check is unreachable: if the context object were null, the method call would already have caused a `NullPointerException` before entering `equals`. A call such as `p1.equals(null)` can safely enter the method when `p1` is not null, and phase 2 returns `false`.

Logical operators and short-circuit evaluation

After the equality discussion, Jackie reviewed `&&` and `||`. The truth tables are familiar from logic, but Java evaluates operands from left to right and may skip the second operand. For `&&`, if the first operand is `false`, the whole expression is already `false`; for `||`, if the first operand is `true`, the whole expression is already `true`. Because of this operational behaviour, these programming operators are not interchangeable with mathematical conjunction and disjunction in every reasoning context.

Using guards to prevent run-time errors

The final examples used short-circuiting as a guard. In `x != 0 && y / x > 2`, the left operand protects the division from being evaluated when `x` is zero. In the `||` version, the corresponding left operand is an error condition such as `x == 0`. Jackie then extended the same idea to array access: check the lower and upper bounds before evaluating `A[i]`. The order matters because putting `A[i]` before its guard can still trigger an array-index error.

Key Terms, Notation, and Structures

- `Object` **class**: The root class that provides inherited methods such as `equals` to every Java class.
- **Inheritance**: The relationship where methods declared in a parent or superclass are available in a child or subclass.
- `extends`: The Java keyword used to express a child-parent class relationship; this lecture previewed it before the full inheritance topic.
- **Default** `equals`: The inherited `Object.equals(Object obj)` version, whose behaviour is reference comparison through `this == obj`.
- **Overriding or redefining**: Declaring a method in a child class with the intended same method signature so that the child-specific version replaces the inherited one for objects of that dynamic type.
- **Dynamic type**: The runtime class of the object currently pointed to by a reference variable; for `x.equals(y)`, the dynamic type of `x` determines which method version is invoked.
- **Context object**: The object before the dot in a method call, such as `x` in `x.equals(y)`.
- **Argument**: The expression passed at the call site, such as `y` in `x.equals(y)`.

- **Parameter**: The variable declared in the method header, such as `obj` in `equals(Object obj)`.
- **Call by value**: Java's method-call mechanism; in an `equals` call, the parameter `obj` receives a copy of the argument reference.
- `this`: Inside an instance method, the reference to the context object on which the method was invoked.
- `==`: For primitive values, it compares values; for compatible reference types, it compares addresses.
- `.equals(...)`: A method call whose object-comparison behaviour depends on the method version selected at runtime.
- **Compilation error**: An error detected before running the program; for example, explicitly writing `p1 == s` for incompatible reference types is not allowed.
- `NullPointerException`: A run-time error that occurs if an instance method is invoked on a null context reference.
- **Short-circuit evaluation**: Left-to-right evaluation of `&&` and `||`, where the second operand may be skipped once the final truth value is determined.
- **Guarding condition**: A condition placed before a risky expression in an `&&` chain so the risky expression is evaluated only when safe.
- **Error condition**: A condition placed before a risky expression in a `||` chain so the risky expression is skipped when an error case has already been detected.

Key Examples and Demonstrations

Default `equals` in `PointV1`

Setup: `PointV1` has fields and a constructor but no explicitly declared `equals`. The tester creates two `PointV1` objects with the same coordinates and calls `p1.equals(p2)`.

Main point: The call compiles because `equals` is inherited from `Object`. Since `PointV1` does not override it, the default version compares references, not point contents.

What to practise: In Eclipse, try stepping into `p1.equals(p2)`. If needed, ask for help attaching the JDK source so you can see the default `Object.equals` implementation.

Choosing the invoked version by dynamic type

Setup: The notes show the general call `x.equals(y)` and the decision rule based on the dynamic type of `x`. A reference variable can point first to a `PointV1` object and later to a `PointV2` object.

Main point: The context object controls dispatch. If `x` currently points to a `PointV1`, the default inherited version is used; if it currently points to a `PointV2`, the overridden version is used.

What to practise: For each `equals` call in the notes, first draw the object box pointed to by the context object, label its dynamic type, then decide whether the default or overridden method body is invoked.

`p1.equals(s)` versus `p1 == s`

Setup: The lecture compared a point reference `p1` and a string reference `s`.

Main point: Explicitly writing `p1 == s` is a compilation error because the two references have incompatible types. In contrast, `p1.equals(s)` compiles because `equals` takes an `Object` parameter; if the default version is used, it still behaves like a reference comparison behind the scenes.

What to practise: Write both expressions in Eclipse and observe the compile-time difference. Then explain why the method call can compile even though the direct `==` expression cannot.

Tracing call by value in `PointV2.equals`

Setup: In a call such as `p2.equals(p3)`, the context object `p2` has dynamic type `PointV2`, so the overridden `equals(Object obj)` method is invoked.

Main point: Before the first line of the overridden method runs, call by value copies the argument reference `p3` into `obj`. Inside the method, `this` refers to `p2`; `obj` refers to the object passed as the argument.

What to practise: Redraw the diagram from the notes with `this`, `obj`, `p2`, and `p3`. Then trace which object each reference points to before evaluating each phase.

Phase 1 and Phase 2 of overridden `equals`

Setup: The overridden `equals` method first checks `this == obj`, then checks `obj == null`.

Main point: Phase 1 returns `true` immediately for self-comparison or aliases. Phase 2 returns `false` immediately when the argument is null. Jackie used this to explain why `this == null` is not a useful internal check: a null context object prevents the method call from starting.

What to practise: Trace `p1.equals(p1)`, an alias case such as `p1.equals(p4)` where both references point to the same object, and `p1.equals(null)`. For each one, identify which phase determines the answer.

Short-circuit guard for division

Setup: The risky expression `y / x > 2` may cause division by zero when `x` is 0.

Main point: The condition `x != 0 && y / x > 2` is safe because the division is skipped when `x != 0` is false. The corresponding `||` version uses an error condition such as `x == 0` to skip the risky expression when the error case is already known.

What to practise: Evaluate both versions with `x = 0, y = 10` and with `x = 5, y = 10`. Record exactly which operand is evaluated and which one is skipped.

Short-circuit guard for array access

Setup: The notes assume an array `A` of length 10, so valid indices are 0 through 9. The expression `A[i] == value` is risky unless `i` is already known to be in range.

Main point: A safe `&&` expression places both guards before the array access, such as checking the lower bound and upper bound before evaluating `A[i]`. Putting `A[i]` between or before its guards may still trigger an out-of-bounds exception.

What to practise: Rewrite the safe guarded condition using `||` instead of `&&`. Make sure the error conditions appear before the risky array access.

Common Pitfalls and Misconceptions

Assuming `equals` must be explicitly declared to compile

Pitfall: Believing that `p1.equals(p2)` cannot compile unless `equals` appears directly in the `PointV1` source code.

Why it causes problems: This misses inheritance from `Object`. The call can compile, but it may not behave as content equality.

How to avoid it: When a method call compiles even though you do not see the method in the class, ask whether it is inherited from a parent class.

Assuming `equals` always means content equality

Pitfall: Treating `p1.equals(p2)` as automatically comparing fields such as `x` and `y`.

Why it causes problems: If the class does not override `equals`, the inherited default version compares references just like ==.

How to avoid it: Before interpreting an `equals` result, identify which method version is invoked: default inherited version or overridden class-specific version.

Looking at the argument instead of the context object

Pitfall: Deciding which `equals` version is called by looking mainly at `y` in `x.equals(y)`.

Why it causes problems: Java dispatches the method by the dynamic type of the context object `x`. The argument affects what the method compares, but not which method body is selected.

How to avoid it: Always start with the object before the dot. Draw the object it points to, label its dynamic type, and then check that class for an overridden `equals`.

Confusing direct == with default `equals`

Pitfall: Thinking that because default `equals` boils down to ==, explicitly writing the same == expression must always compile.

Why it causes problems: Direct == on incompatible reference types can be rejected at compile time, while `equals(Object obj)` can accept any object reference.

How to avoid it: Separate source-level syntax from run-time behaviour. `p1.equals(s)` is a method call with an `Object` parameter; `p1 == s` is a direct operator expression subject to stricter compile-time type rules.

Checking `this == null` inside an instance method

Pitfall: Adding a phase inside `equals` to ask whether `this` is null.

Why it causes problems: That situation is not reachable inside the method body. If the context object were null, Java would throw a `NullPointerException` before entering the method.

How to avoid it: Check `obj == null`, not `this == null`. Distinguish a null argument from a null context object.

Treating `&&` and `||` as freely commutative

Pitfall: Reordering operands as if programming logical operators behave exactly like mathematical conjunction and disjunction.

Why it causes problems: Java evaluates left to right and may skip the second operand. Reordering can change whether a risky expression is evaluated and can introduce run-time errors.

How to avoid it: When an operand can fail, such as division or array access, place the guard before the risky expression.

Placing an array access before all of its guards

Pitfall: Writing a condition where `A[i]` is evaluated before checking both the lower and upper bounds on `i`.

Why it causes problems: For an out-of-range index, Java may evaluate the array access before the missing guard has a chance to stop it.

How to avoid it: Put all needed guard conditions first, for example lower-bound check and upper-bound check, then evaluate `A[i]` only after both have passed.

What You Should Be Able to Do

After this lecture, you should be able to:

- Explain why `p1.equals(p2)` can compile even when `equals` is not explicitly declared in `PointV1`.
- State the behaviour of the default `Object.equals(Object obj)` method.
- Distinguish a default inherited `equals` method from an overridden class-specific `equals` method.

- Given `x.equals(y)`, identify the dynamic type of `x` and determine which `equals` version is invoked.
- Explain why the argument `y` does not determine the invoked `equals` version.
- Trace call by value inside `equals(Object obj)` and identify what `this` and `obj` refer to.
- Justify why `this == obj` is a valid first phase of an overridden `equals` method.
- Justify why `obj == null` should return `false`, and why checking `this == null` inside the method is unreachable.
- Predict the compile-time and run-time behaviour difference between expressions such as `p1 == s` and `p1.equals(s)`.
- Define short-circuit evaluation for `&&` and `||` and explain why operand order matters.
- Construct guarded boolean expressions that avoid division-by-zero and array-index errors.
- Identify incorrect guard orderings and explain what input value exposes the error.
- Use Jackie's examples as preparation for test and exam questions on dynamic dispatch, `equals`, and short-circuit evaluation.

Suggested Practice

- Recreate the `PointV1` example and use the debugger to step into `p1.equals(p2);` ask Jackie for help if Eclipse needs JDK source attachment.
- For every `equals` call in the notes, write three lines: context object, dynamic type, and invoked version of `equals`.
- Draw the `PointV1` memory diagrams for `p1`, `p2`, `p3`, and `s`; then predict the results of the printed expressions before checking in Eclipse.
- Test the compilation difference between `p1 == s` and `p1.equals(s)` using a point object and a string object.
- Trace `p1.equals(p1)`, `p1.equals(null)`, and `p2.equals(p3)` for the `PointV2` overridden version, stopping as soon as a phase returns.
- Write the first two phases of `equals(Object obj)` from memory and explain in words why each one is placed before the later type and content checks.
- Evaluate `x != 0 && y / x > 2` and its `||` counterpart for `x = 0, y = 10` and `x = 5, y = 10`; explicitly mark skipped operands.

- For an array `A` of length 10, write a safe condition for checking whether `A[i]` equals a value. Then intentionally write an unsafe ordering and identify a witness value of `i` that causes the error.
- Review the short-circuit evaluation examples as Written Test 1 practice, since Jackie identified this as test-relevant reasoning.

Where to Review

- **iPad/PDF notes, pages 1–3:** Review the lecture topic, the `PointV1` first example, and the inheritance diagram showing `Object`, `PointV1`, and `PointV2`.
- **iPad/PDF notes, pages 4–5:** Review the dynamic-dispatch principle for `x.equals(y)` and the examples where the same reference variable can point to objects of different dynamic types.
- **iPad/PDF notes, pages 6–7:** Revisit the default `equals` example and the distinction between `p1.equals(s)` and `p1 == s`.
- **iPad/PDF notes, pages 8–12:** Review call by value inside the overridden `equals` method in class `PointV2`, especially the roles of `this`, `obj`, phase 1, and phase 2.
- **iPad/PDF notes, pages 13–19:** Review short-circuit evaluation for `&&` and `||`, the division guard, the array-index guard, and the common ordering errors.
- **Lecture recording:** Rewatch the Eclipse stepping demonstration and Jackie's explanation of why `this == null` is unreachable inside `equals`.
- **Posted source code:** Experiment with `PointV1`, `PointV2`, and the short-circuit examples in Eclipse rather than only reading the notes.
- **Upcoming practice materials:** Use the short-circuit evaluation material when preparing for Written Test 1, and keep the `equals` dispatch rule in mind for test and exam tracing questions.

Closing Reminder

This digest highlights the reasoning path from inherited `equals`, to dynamic method dispatch, to the first phases of an overridden `equals`, and finally to short-circuit guards. It is not enough to memorize the slogans. Revisit Jackie's diagrams, trace the examples by hand, run small variants in Eclipse, and explain why each result happens. In particular, practise deciding the invoked `equals` version from the context object's dynamic type, and practise placing guards before risky expressions.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 13, June 23

Static Types, Dynamic Types, Type Casting, and `equals` method: Phase 3

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 3

Key Examples and Demonstrations 4

Common Pitfalls and Misconceptions 5

What You Should Be Able to Do 7

Suggested Practice 7

Where to Review 8

Closing Reminder 8

Main Ideas

- Lecture 13 resumed object equality by building the type-system ideas needed for Phase 3 of an overridden `equals` method and for the Phase 4 downcast preview.
- A reference variable's **static type** is its declared type, never changes, and determines which method calls the compiler accepts.
- A reference variable's **dynamic type** is the class of the object it currently points to; it can change after assignment and determines which method version runs at runtime.
- Every class inherits methods from `Object`; therefore, inherited methods such as `equals` belong in the set of methods expected by a variable's static type.
- Aliasing does not override compile-time rules: two variables can point to the same object while allowing different method calls because their static types differ.
- A type-cast expression such as `(C1) o3` creates an alias expression with static type `C1`; it does not create a new object or change `o3`'s declared type.
- Phase 3 of `equals`, `this.getClass() != obj.getClass()`, rejects objects of different dynamic types before any attribute comparison.
- Jackie connected static types, dynamic types, casting, and Phases 1–3 of `equals` directly to Written Test 1 preparation.

Roadmap

Second-half roadmap and immediate priorities

Jackie located this lecture at the start of the course's more advanced material: finish `equals`, then study inheritance, polymorphism, dynamic binding, casting, and finally recursion. Lab 2 was due Wednesday, June 24. Written Test 1 was scheduled for Monday, June 29 on eClass, using a mobile phone; Jackie stated that the scope includes material through this lecture, including Phases 1–3 of `equals`, static versus dynamic types, and casting, but not the Thursday lecture. Jackie also introduced the posted lecture digests as optional study roadmaps rather than replacements for the notes or recordings.

Static type and the compiler's expectation set

The lecture used two small classes: `C1` declares `m1()`, `C2` declares `m2()`, and both inherit `equals` from `Object`. A variable's static type is the type in its declaration, such as `C1` in `C1 o1` or `Object` in `Object o3`. To decide whether a call compiles, list the methods declared in that static type together with inherited methods. Thus `o1.m1()` and `o1.equals(...)` compile, while `o1.m2()` does not; an `Object`-typed variable can use `equals` but not `m1()` or `m2()`.

Dynamic type, reassignment, and `getClass()`

The dynamic type is the class of the object currently reached by a reference. It is found by tracing assignments and reading the title of the object box, not by rereading the variable declaration. In the sequence `Object o3;`, `o3 = o1;`, `o3 = o2;`, the static type of `o3` remains `Object`, while its dynamic type changes from undefined/null to `C1`, then to `C2`. The method `getClass()` reveals the current dynamic type, but calling it on a null reference causes a `NullPointerException`.

Why aliasing is not enough, and what a cast changes

After `o3 = o1`, `o1` and `o3` alias the same `C1` object. Nevertheless, `o3.m1()` is still a compilation error because the compiler checks `o3`'s static type, `Object`, rather than tracing its runtime arrow. The cast expression `(C1) o3` provides a new alias expression whose static type is `C1`; therefore `((C1) o3).m1()` compiles. The cast target controls which methods are available to the expression: casting to `C1` supports `m1()`, while casting to `C2` supports `m2()`.

Phase 3 of overridden `equals`

Phase 3 is reached only after Phase 1 has established `this != obj` and Phase 2 has established `obj != null`. It compares the dynamic types of the context object and parameter object:

```
if (this.getClass() != obj.getClass()) { return false; }
```

If the two dynamic types differ, the objects are trivially not equal and Phase 4 is bypassed. In the trace of `p1.equals(s)`, call by value makes `obj` an alias of the `String` argument `s`; `this` refers to the `PointV2` context object `p1`. Their dynamic types differ, so Phase 3 returns `false`.

Preview of Phase 4

Once Phase 3 confirms that the objects have the same dynamic type, Phase 4 will compare their class-specific attributes. The parameter `obj` still has static type `Object`, so direct code such as `obj.x` does not compile. The required pattern begins with a downcast such as `PointV2 other = (PointV2) obj`, after which `other.x` and `other.y` can be accessed. Jackie deferred the detailed Phase 4 rules to the next lecture.

Key Terms, Notation, and Structures

- **Static type (declared type):** The type written in a reference-variable declaration, such as `Object` in `Object o3`; it is fixed after declaration and determines compile-time method availability.
- **Expectation set:** The methods a reference may call at compile time: methods declared in its static type plus methods inherited into that type.

- **Dynamic type:** The class of the object currently referenced at runtime; it can change when the variable is assigned another object's address.
- **Aliasing:** Multiple reference variables or expressions store the same address and point to the same object.
- `Object`: Java's root parent class in this lecture's hierarchy; its default `equals` method is inherited by `C1`, `C2`, and other classes.
- `getClass()`: An inherited instance method used here to obtain the dynamic type of its non-null context object.
- **Type cast / cast expression:** An expression such as `(C1) o3`. It points to the same object as `o3`, but its static type is the cast type `C1`.
- **Compilation error versus runtime exception:** A call outside a receiver's expectation set is rejected before execution; calling an instance method on `null` causes a runtime `NullPointerException`.
- `this`: The context object of an instance-method call. In `p1.equals(s)`, `this` refers to the object reached by `p1`.
- `obj`: The `Object`-typed parameter of `equals`; call by value copies the argument reference into this parameter.
- **Phase 3 of `equals`:** `this.getClass() != obj.getClass()` detects different dynamic types and returns `false` without comparing attributes.
- **Downcast for Phase 4:** A conversion such as `PointV2 other = (PointV2) obj`, needed because `Object` itself has no `x` or `y` fields.

Key Examples and Demonstrations

The `C1`/`C2`/`Object` expectation table

Setup: `C1` declares `m1()`, `C2` declares `m2()`, and both inherit `equals` from `Object`. The reference variables have static types `C1`, `C2`, and `Object`.

Main point: Decide whether each call compiles from the receiver's static type and inherited methods, not from any later runtime diagram. This produces the central contrast: `o1.m1()` compiles but `o1.m2()` does not; `o3.equals(...)` compiles but `o3.m1()` and `o3.m2()` do not.

What to practise: Build the three expectation sets yourself and mark each of the nine calls as "compiles" or "compilation error" before checking the notes.

Tracking `o3`'s static and dynamic types

Setup: Trace `Object o3;`, then `o3 = o1;`, then `o3 = o2;`, where `o1` points to a `C1` object and `o2` points to a `C2` object.

Main point: `o3`'s static type stays `Object` throughout. Its dynamic type evolves from undefined/null to `C1`, then to `C2`. `o3.getClass()` can report the latter two dynamic types, but cannot be called while `o3` is null.

What to practise: Draw the object boxes and arrows after each line. Label static type and dynamic type separately; do not merge them into one label.

Casting as a same-object, different-static-type alias

Setup: After `o3 = o1`, `o1` and `o3` point to the same `C1` object, but `o3.m1()` fails to compile.

Main point: `(C1) o3` is a cast expression that aliases the same object while presenting static type `C1`. Therefore `((C1) o3).m1()` compiles, while `((C1) o3).m2()` does not. Repeating the idea with `(C2) o3` flips the compile-time method availability.

What to practise: For each cast expression, first write its static type, then list its expectation set, and only then decide whether the method call compiles.

Tracing Phase 3 with `p1.equals(s)`

Setup: `p1` points to a `PointV2` object and `s` points to a `String` object. The comparison calls `equals` on `p1` with `s` as the argument.

Main point: The context object makes `this` refer to the `PointV2` object, and call by value makes `obj` refer to the `String` object. Phases 1 and 2 do not return; Phase 3 compares `PointV2` with `String`, finds different dynamic types, and returns `false`.

What to practise: Recreate the trace with an object diagram and explicitly state the result of each of the first three phase conditions.

Common Pitfalls and Misconceptions

Using the dynamic type to decide what compiles

Pitfall: Reasoning that `o3.m1()` should compile merely because `o3` currently points to a `C1` object.

Why it causes problems: The compiler uses only `o3`'s static type, `Object`; it does not trace the runtime arrows.

How to avoid it: For every compilation question, start at the declaration of the receiver variable and build its expectation set.

Treating static and dynamic types as one property

Pitfall: Believing that assigning `o1` or `o2` to `o3` changes `o3`'s declared type.

Why it causes problems: It mixes a fixed compile-time property with a runtime property that can change, leading to incorrect traces and method-call decisions.

How to avoid it: Write static type once from the declaration; update only dynamic type as you trace assignments.

Assuming aliases automatically have the same usable methods

Pitfall: Assuming that two aliases to the same object must allow exactly the same calls.

Why it causes problems: Aliases can have different static types, so their compile-time expectation sets can differ even though they reach the same object.

How to avoid it: Check the static type of the particular expression to the left of the dot, not just the object it eventually reaches.

Misunderstanding a cast

Pitfall: Thinking that `(C1) o3` creates a new `C1` object or permanently changes `o3`'s static type.

Why it causes problems: It leads to incorrect object diagrams and hides the purpose of casting in Phase 4.

How to avoid it: Draw a new alias expression pointing to the same object, label its static type with the cast type, and leave `o3`'s declaration unchanged.

Calling `getClass()` on `null`

Pitfall: Asking a null reference for its dynamic type.

Why it causes problems: There is no context object for the method call, so Java throws a `NullPointerException`.

How to avoid it: Establish that the reference is non-null before calling an instance method such as `getClass()`.

Skipping the type guard in `equals`

Pitfall: Moving directly to attribute comparison without first confirming compatible dynamic types.

Why it causes problems: Objects of different types are already not equal for this design, and the Phase 4 attribute access requires a type-specific view of `obj`.

How to avoid it: Keep the phase order: self-comparison, null check, dynamic-type comparison, then the Phase 4 downcast and content comparison.

Trying to use `obj.x` directly

Pitfall: Accessing class-specific fields through `obj` in `equals`.

Why it causes problems: The parameter's static type is `Object`, whose expectation does not include `x` or `y`.

How to avoid it: After Phase 3 confirms the needed type, downcast `obj` to the class whose fields you need to compare.

What You Should Be Able to Do

After this lecture, you should be able to:

- identify a reference variable's static type from its declaration;
- construct the expectation set for a static type by including declared and inherited methods;
- decide whether a method call compiles from the static type of the expression to the left of the dot;
- distinguish a compilation error from a runtime `NullPointerException`;
- trace the dynamic type of a reference across declarations and assignments;
- explain what `getClass()` reveals and why it cannot be called on a null reference;
- draw an alias diagram that distinguishes static type from dynamic type;
- explain why two aliases to the same object can have different method-call availability;
- explain what a cast expression changes and what it does not change;
- determine which calls compile after casting to `C1` or `C2`;
- trace Phase 3 of an overridden `equals` method, including `this`, `obj`, call by value, and the `getClass()` comparison;
- justify why a downcast is needed before Phase 4 can access class-specific attributes; and
- prepare for Written Test 1 by reviewing Phases 1–3 of `equals`, static versus dynamic types, and casting.

Suggested Practice

- Review Lecture 12's Phase 1 and Phase 2 checks before treating this lecture as a standalone topic; Phase 3 depends on knowing why execution reached it.
- Rebuild the `C1/C2/Object` expectation table without the notes. For every cell, state the receiver's static type and why the call compiles or fails to compile.
- Draw three snapshots for `o3`: after declaration, after `o3 = o1`, and after `o3 = o2`. Record both static and dynamic type in each snapshot.
- For `(C1) o3` and `(C2) o3`, write the static type of the cast expression and decide independently which of `m1()` and `m2()` compiles.
- Explain in writing why `o1.m1()` can compile while `o3.m1()` does not, even when `o1` and `o3` alias the same `C1` object.

- Trace `p1.equals(s)` using a diagram. Mark `this`, `obj`, their dynamic types, the outcome of Phases 1–3, and the point where the method returns.
- Before the next lecture, predict why `obj.x` will not compile in Phase 4 and how `PointV2 other = (PointV2) obj` changes the available fields.
- Submit the best available Lab 2 attempt by Wednesday, June 24, and use the test guide, notes, recordings, and active tracing to prepare for Written Test 1 on Monday, June 29.

Where to Review

- **Lecture 13 iPad/PDF notes, page 3:** definitions and diagrams for static type, dynamic type, aliasing, and cast expressions.
- **Lecture 13 iPad/PDF notes, page 4:** the `Object`–`C1`/`C2` hierarchy and the expectation table used for compilation decisions.
- **Lecture 13 iPad/PDF notes, page 5:** line-by-line tracking of `o3`'s fixed static type and changing dynamic type, including `getClass()`.
- **Lecture 13 notes, pages 6–7:** Phase 3 and the `p1.equals(s)` trace.
- **Lecture 12 notes and recording:** revisit Phases 1–2 of `equals`, especially self-comparison and the null-argument check.
- **Lecture 13 recording:** return to the static/dynamic/casting segment when any compile-time versus runtime distinction remains unclear, and to the final Phase 3 trace before Written Test 1.
- **Posted materials:** use the Written Test 1 guide and practice questions together with the relevant lecture notes; use the Lab 2 instructions for the immediate submission work.

Closing Reminder

The key habit from this lecture is to separate questions in the right order: first determine what compiles from static type, then trace dynamic type and aliases at runtime, and finally trace the selected method body. Revisit Jackie's notes, diagrams, recording, Lab 2 materials, and Written Test 1 practice; draw the arrows, label both types, and explain each decision in your own words. This digest is only a roadmap, not a replacement for that active work.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 14, June 25

Implementing `equals`: Phase 4, JUnit: `assertSame` vs. `assertEquals`

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 4

Common Pitfalls and Misconceptions 5

What You Should Be Able to Do 7

Suggested Practice 7

Where to Review 8

Closing Reminder 9

Main Ideas

- Every method call should be analyzed in two stages: the **static type** of the context object determines whether the call compiles, while its **dynamic type** determines which method version runs at runtime.
- Phase 4 completes the overridden `equals` pattern: after earlier phases establish that two objects are distinct, non-null, and of the same dynamic type, cast the parameter to the specific class and compare the relevant attributes.
- A cast creates another reference (an **alias**) to an existing object with a more specific static type; it does not create a new object and does not change the original reference's static type.
- The phase-2 null check and phase-3 class check can be combined safely only by putting the null test first in a short-circuit `||` condition.
- In JUnit, `assertSame` checks whether two expressions refer to the same object, whereas `assertEquals` behaves differently for primitive values and reference values.
- For reference values, the first argument of `assertEquals` becomes the context object of an `equals` call; therefore, argument order can change which `equals` implementation is invoked and why an assertion fails.
- For Written Test 1, Jackie emphasized reviewing static and dynamic types, type casting, phases 1–3 of `equals`, and short-circuit evaluation. Phase 4 was shown for later work but was not part of that test's stated scope.

Roadmap

1. Reusable method-call rule: compile time versus runtime

Jackie began by separating two questions that must not be mixed. For a call of the form `obj.m(...)`, first ask whether it compiles by looking only at the static (declared) type of `obj`; `m` must be declared in, or inherited by, that type. Only after the code compiles do you trace the program to identify the dynamic type of `obj`, which selects the version of `m` that executes. This distinction is central to the coming inheritance material as well as to choosing an `equals` version.

2. Phase 4 of an overridden `equals` method

The lecture completed the four-phase equality pattern for `PointV2`. Reaching phase 4 means that phase 1 did not return `true`, phase 2 did not return `false`, and phase 3 did not return `false`. Thus, `this` and `obj` are different objects, `obj` is not `null`, and both objects have the same dynamic type. These earlier guards justify the cast

```
PointV2 other = (PointV2) obj;
```

and permit field-by-field comparison using `other.x` and `other.y`.

3. Why the cast is necessary

Although phase 3 establishes that `obj` points to a `PointV2` object, its static type remains `Object`, because it was declared as `Object obj`. Therefore `obj.x` and `obj.y` do not compile: `Object` does not declare those fields. The cast produces an alias whose static type is `PointV2`; the compiler can then accept `other.x` and `other.y`. No new object is allocated.

4. Tracing the complete comparison and selecting the version

For a call such as `p1.equals(p2)` where `p1` and `p2` are distinct `PointV2` objects with matching coordinates, the overridden `PointV2.equals` method is selected because the context object `p1` has dynamic type `PointV2`. The call reaches phase 4, creates `other` as an alias of `p2`, compares both coordinates, and returns `true`. A comparison with different coordinate values reaches phase 4 but returns `false`.

Jackie then contrasted this with `PointV1`, which does not override `equals`. If the context object has dynamic type `PointV1`, Java finds no local `equals` implementation and uses `Object.equals`, which compares references rather than fields.

5. Combining the null and class checks safely

The two early `return false` cases can be merged as

```
if ((obj == null) || (this.getClass() != obj.getClass())) { return false; }
```

This works because `||` evaluates from left to right and skips its right operand when the left operand is `true`. The null check must be on the left: otherwise `obj.getClass()` could be evaluated on `null` and throw a `NullPointerException`.

6. JUnit identity equality versus `equals`

`assertSame(exp1, exp2)` checks whether two expressions store the same address. In contrast, for primitive values, `assertEquals(a, b)` checks their values; for reference values, use the lecture's tracing model

`assertEquals(obj1, obj2)` ≈ `assertTrue(obj1.equals(obj2))`.

The first reference is therefore the context object. When objects use different `equals` implementations, reversing the assertion arguments can change the comparison path. The lecture's mixed `PointV1/PointV2` examples showed that assertions may all fail but for different reasons: reference mismatch under the default method, or dynamic-type mismatch in phase 3 of the overridden method.

7. Practice material and assessment reminders

Jackie pointed students to the posted object-equality traces: the `PointV1/PointV2` assertions and the Lab-3-related `Person` and `PersonCollector` examples. For Written Test 1, bring a mobile phone, verify the required course-system logins in advance, and consult the posted test guide. Jackie also announced an optional Zoom Q&A session at 4 PM on Friday, June 26.

Key Terms, Notation, and Structures

- **Static type (ST):** The type written in a reference variable's declaration. It determines which field and method uses the compiler will allow.
- **Dynamic type (DT):** The runtime class of the object currently referenced. It determines which overridden method version runs after a call has compiled.
- **Context object:** The receiver to the left of the dot in a method call, such as `p1` in `p1.equals(p2)`. Its dynamic type controls dispatch.
- **Phase 4 of `equals`:** After the identity, null, and same-class guards have passed, cast the `Object` parameter to the class being compared and then compare the relevant attributes.
- **Type cast:** An expression such as `(PointV2) obj` that refers to the same object as `obj` but has static type `PointV2` for compilation purposes.
- **Alias:** A different reference variable or expression that stores the same address and points to the same object. `obj` and `other` are aliases after the phase-4 assignment.
- **Default `equals`:** The inherited method from `Object`; it conceptually returns whether `this == obj`, so it checks reference identity.
- **Overridden `equals`:** A class-specific replacement for the inherited version, such as the `PointV2` method that compares coordinates after completing its guards.
- **Short-circuit `||`:** Evaluate left to right; if the left operand is `true`, Java skips the right operand. This permits a null guard before a potentially unsafe dereference.
- `assertSame(exp1, exp2)`: Passes exactly when `exp1` and `exp2` refer to the same object.
- `assertEquals`: For primitive inputs, compare values. For reference inputs in this lecture's model, trace `obj1.equals(obj2)`, making the first argument the context object.
- **Assertion outcome versus boolean value:** A boolean expression evaluates to `true` or `false`; an assertion passes or fails according to that value.

Key Examples and Demonstrations

Phase 4: two distinct but equal `PointV2` objects

Revisit the trace in which `p1` and `p2` are separate `PointV2` objects with equal `x` and `y`. Identify `p1` as `this` and use call by value to make `obj` refer to `p2`'s object. After phases 1–3 are false, draw `other` as an alias of `obj`; then compare the two coordinate pairs. The essential point is that the cast enables legal field access without creating another object.

Same code shape, different dispatch: `PointV1` versus `PointV2`

Compare calls whose context object has dynamic type `PointV1` with calls whose context object has dynamic type `PointV2`. For `PointV1`, follow superclass lookup to the default `Object.equals` and compare addresses. For `PointV2`, execute the overridden four-phase method and compare fields only after the guards permit it. Use the context object's dynamic type, not its argument's type, to choose the version.

A safe one-line null/class guard

Starting from the separate checks `obj == null` and `this.getClass() != obj.getClass()`, explain why

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

is safe, while the reversed order is not. Test mentally with `obj == null`; the correct expression must stop before attempting `obj.getClass()`.

`assertSame`: trace addresses only

Trace three references where `p3` receives the address stored in `p1`. Then `assertSame(p1, p3)` passes, while `assertSame(p2, p3)` fails if `p2` points to a distinct object. Do not bring field values or overridden `equals` into this particular assertion.

`assertEquals`: trace the first argument's `equals`

For reference values, first expand the assertion into an `equals` call: the first argument becomes the context object. In the mixed `PointV1`/`PointV2` examples, distinguish whether failure is caused by different addresses under `Object.equals`, by a class mismatch in phase 3, or later by a field mismatch. Reversing the two arguments can select a different implementation.

Assigned `Person` and `PersonCollector` traces

Use the posted exercises to practise equality beyond a two-field point. For `Person`, identify which attributes define equality. For `PersonCollector`, trace the number of persons and the order-by-order comparison of the contained `Person` objects. Consult the posted solutions only after attempting the traces yourself.

Common Pitfalls and Misconceptions

Using dynamic type to decide compilation

Pitfall: Arguing that `obj.x` or `obj.m()` should compile because `obj` happens to point to an object with that member.

Why it causes problems: The compiler checks the static type, not the runtime object diagram. A parameter declared as `Object` does not support direct access to `x` or `y`.

How to avoid it: First identify the static type for compilation. Use a justified cast to obtain a reference with the required static type.

Treating a cast as object creation or a type change to the original variable

Pitfall: Thinking `(PointV2) obj` builds a new `PointV2` object or permanently changes `obj`'s static type.

Why it causes problems: It leads to incorrect object diagrams and confusion about aliasing.

How to avoid it: Draw the cast result and `obj` with arrows to the same object. The cast adds a reference with a different static type; no object is created.

Writing `obj.x` in phase 4

Pitfall: Replacing `other.x` with `obj.x` because phase 3 has already shown that `obj` points to a `PointV2`.

Why it causes problems: `obj.x` is a compile-time error because `obj`'s declared type is `Object`.

How to avoid it: Cast once, store the alias in `other`, and access `other.x` and `other.y`.

Putting the class check before the null check

Pitfall: Using `(this.getClass() != obj.getClass()) || (obj == null)`.

Why it causes problems: If `obj` is `null`, Java attempts `obj.getClass()` first and throws a `NullPointerException`.

How to avoid it: Put `obj == null` first in the `||` expression so short-circuiting protects the right-hand side.

Assuming `assertSame` and `assertEquals` test the same thing

Pitfall: Using `assertSame` for a field-based test, or assuming that `assertEquals` always checks fields.

Why it causes problems: `assertSame` checks addresses only. `assertEquals` on references follows the selected `equals` implementation, which might still be the default address comparison.

How to avoid it: Identify the assertion type, then identify the dynamic type of the first reference passed to `assertEquals`.

Ignoring assertion argument order

Pitfall: Assuming `assertEquals(a, b)` and `assertEquals(b, a)` must trace identically.

Why it causes problems: The first argument is the context object of `equals`; swapping order can choose a different dynamic type and method version.

How to avoid it: Expand each assertion into its `equals` call before tracing it.

What You Should Be Able to Do

After this lecture, you should be able to:

- apply the two-question method-call framework: use static type to decide compilation, then dynamic type to identify the version invoked at runtime;
- explain the guarantees obtained by reaching phase 4 of an overridden `equals` method;
- justify why `PointV2 other = (PointV2) obj;` is safe after phase 3 and draw `obj` and `other` as aliases;
- explain precisely why `obj.x` does not compile but `other.x` does;
- trace a complete `PointV2.equals` call through all four phases and determine whether it returns `true` or `false`;
- determine whether a call uses the inherited default `Object.equals` or an overridden version by tracing the dynamic type of the context object;
- construct and justify the safe combined null/class condition using short-circuit `||`;
- distinguish `assertSame`, `assertTrue`, `assertFalse`, and `assertEquals` in terms of their pass/fail conditions;
- expand a reference-type `assertEquals(obj1, obj2)` into the corresponding `equals` call and identify the context object;
- explain whether an assertion failed because of reference identity, a class mismatch, or a field comparison; and
- locate the posted tracing exercises and use them to practise equality for both simple objects and collections of objects.

Suggested Practice

- Redraw the phase-4 object diagram with `this`, `obj`, and `other`; label each reference's static type and the object's dynamic type.
- Write the four-phase `equals` pattern from memory for a small class with two primitive attributes. Then explain why each earlier phase must precede the field comparison.
- Take several calls of the form `a.equals(b)` and answer in order: Does the call compile? Which `equals` version is selected? Which phase returns, if any?

- Practise the merged guard with both `obj == null` and non-null inputs. Explain why the null check must be the left operand of `||`.
- For three references where two are aliases, predict pass/fail for `assertSame` before running the tests.
- For primitive and reference inputs, expand `assertEquals` into its underlying comparison. For reference inputs, trace the `equals` call from the first argument.
- Complete the posted object-equality tracing sets: first the `PointV1/PointV2` assertions, then the `Person/PersonCollector` traces connected to Lab 3. Attempt each trace before checking its solution.
- For Written Test 1 preparation, review the stated material from Lecture 13: static and dynamic types, type casting, phases 1–3 of `equals`, and short-circuit evaluation. Verify the required course-system logins and review the posted test guide before arriving with your mobile phone.

Where to Review

- **Lecture 14 iPad/PDF notes, page 3:** the two-question method-call framework: static compilation versus dynamic dispatch.
- **Pages 4–7:** phase 4 of `PointV2.equals`, complete traces, and the contrast between default and overridden `equals`.
- **Page 8:** the short-circuit exercise for safely combining the null and dynamic-class checks.
- **Pages 9–11:** `assertSame`, primitive versus reference `assertEquals`, and why the first assertion argument controls the `equals` call.
- **Pages 12–19:** the assigned object-equality exercises, including `Point`, `Person`, and `PersonCollector` cases.
- **Lecture 12 materials:** revisit the short-circuit evaluation discussion before attempting the merged-condition exercise.
- **Lecture site under Object Equality:** use the posted tracing exercises and solutions; connect the `Person/PersonCollector` material to Lab 3.
- **Written Test 1 guide and course syllabus:** review the test procedure and login requirements; the syllabus contains the Zoom link used for the optional Q&A session announced in class.

Closing Reminder

The key habit from this lecture is to keep compile-time and runtime reasoning separate: use static type to decide whether code is legal, then use dynamic type to decide which method body runs. Review Jackie's notes and recording, redraw the alias diagrams, trace the `equals` phases line by line, and run or write JUnit tests rather than relying on intuition alone. This digest is only a roadmap; genuine mastery requires active tracing, coding, debugging, and practice with the posted exercises and Lab 3 materials.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 15, June 30

Equality Reasoning, Design Principles, SMS Example: Design Attempt 1

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 7

Where to Review 8

Closing Reminder 8

Main Ideas

- **Reference equality versus contents equality:** For reference variables, == asks whether two references store the same address, while an overridden `equals` method can compare the relevant state of two objects.
- **A useful equality implication:** If two references are identical, then `equals` must return `true` for the object compared with itself. The converse is false: two different objects can have equal contents.
- **Equality can be composed:** A well-designed `PersonCollector.equals` compares its meaningful size first and then delegates each element comparison to `Person.equals`; it should not compare array or object references as a substitute for contents.
- **Dispatch inside `equals`:** In a composite comparison, identify the receiver from left to right. Each stored `Person` is the receiver, so Java uses `Person.equals`.
- **Inheritance is motivated by design quality:** The Student Management System example exposed problems with one class that stores a numeric `kind` for resident and non-resident students. The design mixes unrelated state and repeats the same kind checks across methods.
- **Two design principles guided the critique:** Cohesion asks whether the members of one class serve a unified purpose. The Single Choice Principle (SCP) asks whether a change affects only the necessary minimum number of locations. Both prepare the motivation for inheritance and polymorphism.

Roadmap

1. Course reminders and Programming Test 2 preparation

Jackie noted that digests through Lecture 14, the Lab 2 solution, the Programming Test 2 guide, and the Practice Test 2 starter project were available. Lab 3 had also been released. Programming Test 2 is scheduled for Monday, July 6 during the regular live session and uses the older test format rather than eClass. Use the guide to confirm coverage, study the Lab 2 and Lab 3 patterns, and attempt the practice project before consulting a solution. Jackie expected to release Programming Test 1 and Written Test 1 results by Friday after final verification.

2. Two meanings of equality and logical implication

For objects, distinguish reference equality, `o1 == o2`, from contents equality, `o1.equals(o2)` when `equals` is properly overridden. The lecture used logical implication to examine two claims: "reference-equal implies contents-equal" and its converse. An implication is false only in the true–false case: the antecedent is true while the consequence is false.

Let `A` mean `o1 == o2` and `B` mean `o1.equals(o2)`. `A` implies `B`: if both references point to the same object, that object is trivially equal to itself. However, `B` does not imply `A`: two distinct `PointV2` objects can have matching fields and therefore be contents-equal even though their addresses differ. Revisit the truth table and the counterexample rather than trying to memorize the conclusion.

3. `Person.equals`: choose the correct comparison for each field

The `Person` example applies the usual `equals` guards before comparing fields. Its numeric fields, such as weight and height, are primitive values and can be compared with `==`. Its first and last names are `String` objects, so the comparison must call `String.equals`. The receiver of `firstName.equals(...)` has dynamic type `String`; Java therefore uses the overridden `String.equals`, which compares character contents rather than addresses.

The notes also revisited the safe combined guard for the null and class cases. The null test must be the left side of the `||` condition so that short-circuit evaluation prevents a call to `obj.getClass()` when `obj` is `null`.

4. `PersonCollector.equals`: equality within equality

A `PersonCollector` stores a `Person` array and a count of stored persons. Two collectors cannot be equal when their counts differ. When the counts agree, compare corresponding persons index by index. The important operation is conceptually

```
persons[i].equals(...).
```

Do not compare the two arrays directly by reference, and do not compare individual `Person` references by `==`. This is an *equals-within-equals* pattern: the composite class reuses the equality definition of the contained class.

5. Entering inheritance: scope and motivation

After completing the object-equality discussion, Jackie introduced inheritance as a major remaining course topic. Upcoming work will use Java constructs such as `extends`, `super`, type casting, and `instanceof`, while the lectures will emphasize the conceptual rules behind polymorphism and dynamic binding. The immediate question was not yet how to write inheritance syntax, but why a design without inheritance becomes difficult to maintain.

6. Student Management System: common and varying responsibilities

The motivating system manages resident students (RS) and non-resident students (NRS). Both have a name, registered courses, a registration operation, and a tuition base amount calculated from course fees. The tuition adjustment differs: resident students use a premium rate and non-resident students use a discount rate. The lecture first considered how to model this without inheritance before introducing a better design later.

7. Two design principles and the first non-inheritance attempt

Cohesion means attributes and methods in one class should serve a unified purpose. **SCP** means a change should affect a minimum necessary number of places; repeated code is a warning sign. The first design used one `Student` class with an integer `kind`, plus both `premiumRate` and `discountRate`. An RS object leaves its discount rate irrelevant, while an NRS object leaves its premium rate irrelevant. The class therefore mixes separate purposes and lacks cohesion.

The same `kind` condition must appear in `getTuition`, `register`, and every other kind-dependent method. Adding an international-student kind, or removing a kind, requires editing every such conditional. Because each method has different branch actions, the repeated decision cannot simply be hidden in one helper method in this design. This is the SCP problem that motivates the later inheritance solution.

Key Terms, Notation, and Structures

- **Reference equality:** For object references, `o1 == o2` is `true` exactly when both variables store the same address.
- **Contents equality:** `o1.equals(o2)` asks whether the relevant state of the two objects is equal according to the version of `equals` selected for the receiver.
- **Logical implication:** $p \Rightarrow q$ is false only when p is true and q is false. The left side is the *antecedent* (or assumption); the right side is the *consequence*.
- **Equality propositions:** `A` denotes `o1 == o2`; `B` denotes `o1.equals(o2)`. The lecture established $A \Rightarrow B$, but not $B \Rightarrow A$.
- **Dynamic dispatch:** At runtime, the dynamic type of the context object, to the left of the dot, selects the applicable overridden method version.
- **`Person.equals`:** Compare primitive measurements with `==`; compare the `String` name fields with `String.equals`. The usual self, null, and dynamic-class guards precede field comparisons.
- **`PersonCollector`:** A composite object with a `Person[]` array and a count `nop` of currently stored persons. Equality requires matching counts and matching persons at corresponding valid indices.
- **Equals-within-equals:** A composite class delegates part of its equality decision to the `equals` methods of its member objects, such as the equality method of each stored `Person`.
- **Cohesion:** Attributes and methods in a single class should serve the same purpose or fit one unifying theme.
- **Single Choice Principle (SCP):** A change should impact a minimum necessary number of locations. Repeated kind-based `if/else if` checks are a sign of an SCP violation.

- **Kind-based Student design:** One `Student` class uses an integer `kind` to represent RS or NRS, while storing both `premiumRate` and `discountRate`. The example deliberately illustrates a non-inheritance design that will be criticized.
- **Base amount, premium rate, discount rate:** Tuition begins with the sum of registered course fees. The adjustment differs by student kind in the first design attempt.

Key Examples and Demonstrations

Reference equality implies contents equality, but not conversely

Take two propositions: `A` is "the references are identical" and `B` is "the objects are contents-equal." If `A` is true, both references denote the same object, so comparing the object with itself makes `B` true. To refute the converse, use two separate `PointV2` objects with the same coordinates: `==` is false, while the overridden `equals` returns `true`. This is the essential counterexample for $B \Rightarrow A$.

Using `String.equals` inside `Person.equals`

Each name field of a `Person` is a `String` reference. Its equality call invokes the overridden `String.equals`, which compares character contents. In contrast, weight and height are primitive values and use `==`.

Comparing two `PersonCollector` objects

The object diagrams showed two collectors, each with an array and a current count. First compare the counts. Only when they match should you compare index `0` to index `0`, index `1` to index `1`, and so on through the stored range. Each element comparison delegates to `Person.equals`; a difference in count or in any corresponding person makes the collectors unequal.

Following the receiver to select an `equals` version

For `persons[i].equals(...)`, read left to right: `this` is a `PersonCollector`; `persons` is a `Person[]` array; indexing selects a `Person`. Thus the context object is a `Person`, and the call uses the `Person` equality implementation. This is a reusable tracing pattern for nested expressions.

Student Management System, Design Attempt 1

The single `Student` class keeps shared course data together with `kind`, `premiumRate`, and `discountRate`. `getTuition` first computes a common base amount, then chooses an adjustment using `kind`; `register` similarly chooses a kind-dependent maximum. The example is useful because it makes the two design failures visible: irrelevant fields for each kind, and duplicated conditionals across methods.

Adding a new student kind

If a third kind, such as an international student, is introduced, each kind-dependent method needs another `else if (this.kind == 3)` branch with its own behavior. Removing a kind requires deleting corresponding branches everywhere. Use this maintenance scenario to test the design against SCP before learning the inheritance-based alternative.

Common Pitfalls and Misconceptions

Treating == and `equals` as interchangeable

For references, == compares addresses. An overridden `equals` method may compare contents. Two distinct but state-equal objects are the standard counterexample: == can be `false` while `equals` is `true`.

Reversing the equality implication

The lecture's valid property is "reference-equal implies contents-equal." Do not infer that contents-equal objects must be the same object. Use the implication truth table and look specifically for the only false case: true antecedent and false consequence.

Using == for `String` contents

A name field is a reference. Use `String.equals` to compare character contents; == would test whether the two variables refer to the same `String` object.

Comparing a collector's arrays or elements only by reference

Two arrays may be different objects even when they store equal persons, and two `Person` objects may be distinct but contents-equal. Compare the count first, then compare corresponding valid elements using `Person.equals`.

Evaluating the class check before the null check

A condition that calls `obj.getClass()` before establishing that `obj` is not `null` can throw a `NullPointerException`. In the combined guard, put `obj == null` on the left of `||`.

Mistaking the numeric `kind` field for a clean OO design

A type code can make a non-inheritance design appear flexible, but it also mixes kind-specific state in one class and forces repeated conditionals. Adding or removing a kind then creates scattered edits. This is exactly what the Cohesion and SCP analysis is meant to reveal.

Preparing for Programming Test 2 by only reading solutions

Jackie emphasized that passive solution viewing does not develop the required programming proficiency. Attempt the Practice Test 2 starter project and review the Lab 2 and Lab 3 patterns actively before checking solutions.

What You Should Be Able to Do

After this lecture, you should be able to:

- distinguish reference equality from contents equality for Java reference variables;
- reproduce the truth conditions for logical implication and explain why only the true–false case makes an implication false;
- justify $A \Rightarrow B$ for `A: o1 == o2` and `B: o1.equals(o2)`, and give a concrete counterexample to $B \Rightarrow A$;
- identify the receiver in a dotted expression and determine which `equals` implementation is invoked by dynamic dispatch;
- explain why `Person.equals` uses `==` for primitive measurements but `String.equals` for name fields;
- trace and implement the main comparison idea for `PersonCollector.equals`: compare the count, then compare corresponding stored persons by contents;
- explain why a combined null/class guard must test for `null` before calling `getClass()`;
- define Cohesion and SCP in your own words and use each principle to critique a class design;
- identify the irrelevant fields and duplicated kind checks in the first `Student` design; and
- explain why adding or removing a student kind creates many edits in the kind-based design.

Suggested Practice

- Recreate the four-row implication truth table. Mark the one false row, then use it to justify why reference equality implies contents equality.
- Draw two separate `PointV2` objects with the same fields. Explain why `p1 == p2` is `false` but `p1.equals(p2)` can be `true`.
- Write or trace a `Person.equals` method. For each field, state whether it is primitive or a reference and which comparison is appropriate.

- Trace `persons[i].equals(...)` from left to right, identifying the type at each step and the final `equals` version called.
- Design test cases for `PersonCollector.equals`: different counts; equal counts with a mismatch at one index; and equal counts with all matching persons. Draw the arrays and references before tracing.
- Complete the posted Object Equality tracing exercises, including the `assertSame` and `assertEquals` tasks, before checking their solutions.
- For the Student Management System attempt, list every method that branches on `kind`. Then simulate adding `kind == 3` and identify all locations that must change.
- Before Programming Test 2 on Monday, July 6, study the posted guide, Lab 2 solution, and Lab 3. Attempt the Practice Test 2 starter project yourself before viewing the solution or attending a review session.

Where to Review

- **Lecture 15 iPad/PDF notes, pages 3–4:** the reference-equality/content-equality property and the implication truth table.
- **Pages 5–8:** the `Person.equals` exercise, the combined null/class guard, and element-wise `PersonCollector.equals` using `Person.equals`.
- **Page 9:** the Student Management System problem statement; identify the shared features and the resident/non-resident differences before studying the designs.
- **Page 10:** the definitions of Cohesion and the Single Choice Principle.
- **Pages 11–13:** the first `Student` design, repeated `kind` tests in `getTuition` and `register`, and the maintenance effect of adding or removing a kind.
- **Lecture site, Object Equality exercises:** revisit the assigned tracing exercises and posted solutions for `assertSame`, `assertEquals`, `Person`, and `PersonCollector`.
- **Programming Test 2 materials:** use the guide, Lab 2 solution, Lab 3, and Practice Test 2 starter project to prepare actively for the July 6 test.

Closing Reminder

This lecture connected two habits that will matter throughout advanced OOP: reason precisely about what equality means, and judge designs by their maintenance consequences. Review Jackie's notes and recording, practise the equality traces and composite `equals` patterns, and use Cohesion and SCP to explain why the kind-based design is fragile. The inheritance solution comes next, but this digest is only a roadmap; build understanding through active tracing, coding, and reflection on the posted exercises and labs.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 16, July 2

SMS: Design Attempt 2, Inheritance: Code Reuse, Visibility

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 7

Where to Review 8

Closing Reminder 8

Main Ideas

- A design can produce correct results yet still be difficult to maintain: Design Attempt 2 separates resident and non-resident students, improving cohesion but duplicating common state and behaviour.
- Inheritance addresses the duplication by placing shared attributes and methods in a parent `Student` class, then letting `ResidentStudent` and `NonResidentStudent` extend it.
- Child classes inherit non-private parent members, may add child-specific features, and may override inherited methods when their behaviour must differ.
- The `super` keyword can reuse immediate-parent logic: `super.method()` for a method version and `super(...)` for a constructor.
- When drawing an object, start from its dynamic type: a child object includes inherited parent attributes as well as its child-specific attributes.
- Java visibility controls which code can use a class, attribute, or method; in particular, `protected` means package access plus subclass access.
- For Programming Test 2 next Monday, July 6, Jackie emphasized active timed practice with Lab 2, Lab 3, and Practice Test 2 rather than only reading solutions.

Roadmap

1. Why Design Attempt 2 is still not enough

Design Attempt 2 uses two unrelated classes: one for resident students and one for non-resident students. This fixes the earlier cohesion problem because a resident-student object has a premium rate only, while a non-resident-student object has a discount rate only. However, both classes repeat the same common state and work: a name, an array of registered courses, a course counter, constructor initialization, registration logic, and most of the base-tuition calculation.

This is a violation of the *Single Choice Principle* (SCP). A new registration constraint or a revised base-tuition formula must be copied into every duplicate implementation. The design can work at run time, but it is not a maintainable structure.

2. The collection problem without inheritance

A student-management system must eventually manage *all* students. With two unrelated student classes, the system needs separate arrays for the two kinds of students. A method that registers every student in a common course must therefore use separate loops.

The problem grows with the number of student kinds: more kinds mean more unrelated classes, arrays, and loops. This is another reason that "it works" is not enough to judge a design.

3. Design Attempt 3: use inheritance for common code

The inheritance design introduces a parent class `Student` and two child classes:

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

Put the common features in `Student`: the student's name, registered-course array, course counter, constructor work for those common fields, registration behaviour, and the common base-tuition calculation. The two child classes inherit these non-private features instead of redeclaring them.

Each child keeps only what is specific to its own purpose. For example, `ResidentStudent` adds `premiumRate`; `NonResidentStudent` adds `discountRate`. This organization aims to satisfy both SCP (shared code is written once) and cohesion (each child owns only its specialized features).

4. Overriding `getTuition` and using `super`

The parent version of `getTuition()` calculates the base tuition by summing course fees. Each child needs a specialized final calculation, so it overrides the method. Rather than rewriting the base calculation, the child calls the inherited parent version:

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

Then the resident-student version applies `premiumRate`; the non-resident-student version applies `discountRate`. This is the central code-reuse pattern from the lecture: preserve common logic in the parent, then add the child-specific step in the overriding method.

Similarly, `super(name)` in a child constructor invokes the immediate parent's constructor so that inherited fields are initialized without copying the parent's constructor body.

5. The scope of `super`

In a hierarchy such as `P`, `C extends P`, and `GC extends C`, `super.m()` inside `C` refers to the version in `P`. Inside `GC`, `super.m()` refers to the version in `C`. Java does not permit `super.super.m()`.

Keep the two roles separate: `super.m()` refers to an immediate parent's method version, while `super(...)` invokes an immediate parent's constructor.

6. Object diagrams and dynamic type

For an object whose dynamic type is `Student`, draw only the attributes declared for `Student`. For an object whose dynamic type is `ResidentStudent`, draw all inherited `Student` attributes *plus* `premiumRate`. For a `NonResidentStudent` object, draw the inherited attributes plus `discountRate`.

This is the same tracing habit used earlier in the course: identify the dynamic type before deciding what object structure and behaviour are relevant.

7. Visibility modifiers in Java

Jackie then introduced four visibility choices for attributes and methods: `private`, no modifier, `protected`, and `public`. For top-level classes, the lecture focused on no modifier and `public`.

A class with no modifier is visible only within its package; a `public` class is visible from other packages. For members, `private` means only the declaring class; no modifier means the same package; `protected` means the same package plus subclasses, including subclasses in other packages; and `public` means accessible everywhere.

Key Terms, Notation, and Structures

- **Cohesion:** Attributes and methods in one class should serve a consistent, unifying purpose.
- **Single Choice Principle (SCP):** A requirement change should affect a minimum number of places; repeated code is a warning sign that SCP is being violated.
- **Inheritance:** A class relationship through which a child class receives non-private attributes and methods from a parent class.
- **Parent class / superclass:** The class to the right of `extends`, such as `Student` in `class ResidentStudent extends Student`.
- **Child class / subclass:** The class to the left of `extends`, such as `ResidentStudent`.
- `extends`: Java syntax for creating a parent-child relationship. A class may name only one superclass after `extends`.
- **Inherited member:** A non-private parent attribute or method that is available in a child class without being redeclared.
- **Method overriding:** A child class declares its own version of an inherited method, such as a child-specific `getTuition()`.
- `super.method()`: Invokes the immediate parent's version of a method, often to reuse common work inside an overriding method.
- `super(...)`: Invokes the immediate parent's constructor to initialize inherited state.
- **Dynamic type:** The run-time type of the object a reference currently points to; use it to determine the full object diagram.
- **Package:** A grouping of related Java classes. Visibility rules distinguish code in the same package from code in other packages.
- `private`: Accessible only inside the declaring class.
- **No modifier (package-private):** Accessible from classes in the same package.

- `protected`: Accessible from classes in the same package and from subclasses, including subclasses in a different package.
- `public`: Accessible from any class.

Key Examples and Demonstrations

Design Attempt 2: two unrelated student classes

Setup: `ResidentStudent` and `NonResidentStudent` each store a name, course array, course count, constructor logic, `register`, and most of `getTuition`. They differ mainly in their rate and final tuition adjustment.

Main point: Separating the rate fields makes the classes more cohesive than Design Attempt 1, but copying the shared code violates SCP. A policy change to registration or base tuition must be made in both places.

What to practise: Mark the duplicated fragments in the two classes. Then imagine adding a maximum-registration constraint or an inflation factor and list every location that would need editing.

Design Attempt 3: parent code plus child-specific tuition

Setup: `Student` contains the common fields and calculates the base tuition. Each child class extends it, adds its own rate field, and overrides `getTuition()`.

Main point: An overriding method calls `super.getTuition()` to reuse the base calculation, then applies its specialized rule. The common code is stored once in the parent.

What to practise: For each child, write the sequence: call the parent version, obtain the base amount, apply the appropriate rate, and return. Explain why rewriting the course-fee loop would reintroduce duplication.

Three levels of overriding

Setup: `P` defines `m`; `C extends P` overrides `m`; `GC extends C` overrides it again.

Main point: From `C`, `super.m()` reaches `P.m()`. From `GC`, `super.m()` reaches `C.m()`. There is no legal `super.super.m()` form.

What to practise: Draw the three-level hierarchy and annotate exactly which method version each legal `super.m()` call reaches.

Visibility across packages and subclasses

Setup: The notes use classes in different packages, including a `Chair` class and subclasses such as `BubbleChair` and `RockingChair`.

Main point: Visibility is not determined only by whether a class is a subclass. Package membership also matters. The important special rule is that `protected` combines package access with subclass access.

What to practise: Make a four-row visibility table for `private`, no modifier, `protected`, and `public`; test each row against a same-class, same-package, different-package subclass, and unrelated-class situation.

Common Pitfalls and Misconceptions

"The two-class design works, so it must be good"

Pitfall: Judging Design Attempt 2 only by whether the tuition output is correct.

Why it causes problems: The design duplicates common code, so later policy changes require multiple edits and may create inconsistent versions.

How to avoid it: Judge designs using both cohesion and SCP. Ask where common code is stored and how many locations a realistic change would affect.

Copying parent logic into every child

Pitfall: Rewriting the common course-fee loop, constructor initialization, or registration method in each subclass.

Why it causes problems: This recreates the duplicates that inheritance is meant to eliminate.

How to avoid it: Put common non-private features in the parent. In an overriding method, first ask whether `super.method()` already provides the common portion.

Redeclaring inherited state

Pitfall: Declaring a second copy of common parent attributes such as name, registered courses, or course count in a child class.

Why it causes problems: Those non-private attributes are already inherited; duplicate declarations make the design confusing and defeat the intended factoring.

How to avoid it: Identify what belongs to every child and declare it once in the parent. Add fields in a child only when they are genuinely child-specific.

Trying to chain `super`

Pitfall: Writing or expecting `super.super.m()` to reach a grandparent's implementation.

Why it causes problems: Java only lets `super` refer to the immediate parent; chaining it is illegal.

How to avoid it: Trace the hierarchy one level at a time and use only the immediate-parent version reached by `super.m()`.

Treating `protected` as "subclasses only"

Pitfall: Forgetting that `protected` also gives access to classes in the same package.

Why it causes problems: You may predict compilation results incorrectly or choose an access level that is broader or narrower than intended.

How to avoid it: Remember the rule: `protected` means package access *plus* subclass access. Contrast it with no modifier, which is package access only.

What You Should Be Able to Do

After this lecture, you should be able to:

- evaluate Design Attempts 1 and 2 using cohesion and the Single Choice Principle;
- identify duplicated fields, constructor code, methods, and loops in unrelated but similar classes;
- explain why separate arrays and loops for each student kind do not scale well;
- write and interpret an inheritance declaration such as `class ResidentStudent extends Student`;
- distinguish parent/superclass from child/subclass in an inheritance diagram;
- decide which attributes and methods belong in a common parent versus in a specialized child;
- explain which non-private parent members are inherited and why they need not be redeclared in the child;
- override a child-specific method and reuse shared logic through `super.method()`;
- use `super(...)` conceptually to invoke the immediate parent's constructor;
- trace a three-level hierarchy and justify why each `super.m()` call reaches only one level upward;
- draw a `Student`, `ResidentStudent`, or `NonResidentStudent` object with the correct inherited and child-specific attributes according to its dynamic type;
- apply the rules for `private`, no modifier, `protected`, and `public` to predict whether an access is allowed.

Suggested Practice

- Revisit Design Attempts 1 and 2, then write a short comparison: which design principle each one violates, and why.
- Redraw Design Attempt 3 with `Student` as the parent and the two student kinds as children. Mark common versus child-specific fields and methods.
- Reimplement the `getTuition()` idea on paper: parent computes the base amount once; each child calls `super.getTuition()` and applies its own rate.

- Draw the `P–C–GC` hierarchy and test yourself on the legal meaning of `super.m()` from each class.
- Download and explore the posted `visibility_classes_vs_attribute_methods` project. For each intended compilation error, state which visibility rule causes it.
- For Programming Test 2 on Monday, July 6, actively redo Lab 2 under a time limit, redo the `equals` work from Lab 3, and attempt Practice Test 2 before checking solutions or walkthrough material.
- Read the Programming Test 2 guide and confirm the stated coverage and programming requirements policy.

Where to Review

- **iPad/PDF notes, pages 3–6:** Design Attempt 2, its duplicated code, maintenance examples, and the collection problem caused by unrelated student types.
- **iPad/PDF notes, pages 7–11:** Design Attempt 3, `extends`, overriding `getTuition`, both forms of `super`, the immediate-parent restriction, and parent/child object diagrams.
- **iPad/PDF notes, pages 12–14:** Class-level and member-level visibility, including the package-plus-subclass rule for `protected`.
- **Lecture recording:** Rewatch the comparison of Design Attempt 2 and Design Attempt 3 if you are unsure why a correct program may still be a poor design.
- **Course site:** Review the Programming Test 2 guide, the released Lab 2 solution, Lab 3, Practice Test 2, and the posted visibility project.

Closing Reminder

Inheritance is not just new Java syntax. Revisit the notes and recording until you can explain why the parent class holds common code, why children add or override only what is specific to them, and how `super` supports reuse without duplication. Then practise actively: trace method calls, draw objects by dynamic type, compile small visibility examples, and work through the relevant labs and practice test. This digest is a roadmap, not a replacement for that work.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 17, July 7

Inheritance: Expectations, Polymorphism, and Dynamic Binding

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- The lecture began by placing the next assessments on the calendar: Lab 3 was due July 8, Lab 4 was to be released July 8, Written Test 2 was scheduled for July 20, Programming Test 3 for July 27, and the final exam was tentatively scheduled for August 10.
- For inheritance programming, Jackie gave a practical modifier rule: start attributes as `private`, then change selected attributes to `protected` only when subclasses need them for code reuse.
- The main technical concept introduced was *expectation*: the set of attributes and methods that may legally appear after the dot of a reference variable.
- Expectation is determined by the reference variable's *static type*, not by the object it currently points to at run time.
- Polymorphism allows a parent-type variable, such as `Student s`, to point to child objects, such as `ResidentStudent` or `NonResidentStudent` objects.
- Dynamic binding says that an overridden method call, such as `s.getTuition()`, runs the version belonging to the current *dynamic type* of the object.
- Multi-level inheritance requires accumulating expectations from a class and all of its ancestors, as illustrated by the smartphone hierarchy.
- Jackie emphasized that static type, dynamic type, expectation, substitution, and dynamic binding are foundational for the rest of the inheritance unit.

Roadmap

1. Logistics and assessment calendar

Jackie announced a tentative final-exam time of Monday, August 10, from 9:00 AM to 12:00 noon. The exam was described as paper-and-pen, with black or blue ink or pencil allowed, and the date, time, and location still subject to Registrar confirmation. Students were told to mark it in pencil for now.

The immediate course priorities were also reviewed: Lab 3 was due the next day, Lab 4 would be released the next day and would be directly relevant to Programming Test 3, Written Test 2 was scheduled for July 20, and Programming Test 3 for July 27. Programming Test 2 grading was in progress.

2. Practical rule for `private` and `protected`

Before entering the main inheritance concepts, Jackie revisited visibility modifiers. The practical advice was not to over-plan every modifier at the beginning. Start by declaring attributes as `private`; then, during development, if a subclass needs an inherited attribute to avoid duplicated declarations or duplicated logic, change that attribute to `protected`.

This connects access control to the Single Choice Principle. The goal is not to expose everything, but to expose only the parent state that should truly be inherited and reused by subclasses.

3. From keywords to foundations: expectation and substitution

The lecture then shifted away from just using keywords such as `extends`, `super`, and later `instanceof`. Jackie introduced two foundational ideas for understanding inheritance: *expectation* and *substitution*. The lecture focused mainly on expectation, polymorphism, and dynamic binding, with the formal substitution rule prepared for the next class.

4. Expectation: what can appear after the dot

Expectation means the range of attributes and methods that can legally be called on a reference variable. In code form, when writing

```
obj.
```

the expectation of `obj` is the set of names that may appear after the dot without causing a compilation error.

The crucial rule is that expectation is determined only by static type. For example, if `s` is declared with static type `Student`, then `s` may use expectations from `Student`, even if at run time it later points to a `ResidentStudent` object.

5. Student hierarchy expectation table

The notes used the hierarchy `Student`, `ResidentStudent`, and `NonResidentStudent`. Common expectations, such as `name`, `registeredCourses`, `numberOfCourses`, `register`, and `getTuition`, come from `Student` and are inherited by both child classes.

Child-specific expectations stay with the child side of the hierarchy. `premiumRate` and `setPremiumRate` belong to `ResidentStudent`; `discountRate` and `setDiscountRate` belong to `NonResidentStudent`. A sibling's feature is not part of the other sibling's expectation.

6. Polymorphism: why `s = rs` and `rs = s` differ

Jackie compared two assignments between related types: assigning a child reference to a parent variable and assigning a parent reference to a child variable. The valid direction is upward in the hierarchy: a `ResidentStudent` object may be used where a `Student` is expected.

The invalid direction was explained by contradiction. If `rs = s` were allowed, then a `ResidentStudent` variable could point to a plain `Student` object. Since `rs`'s static type includes resident-specific expectations such as `premiumRate`, code using `rs.premiumRate` would compile, but the actual `Student` object would not contain that field. The compiler prevents this unsafe situation by rejecting the assignment.

7. Dynamic binding: same static type, changing dynamic type

The dynamic-binding trace used variables `s`, `rs`, and `nrs`. The static type of `s` remains `Student` throughout the trace. However, after `s = rs`, the dynamic type of the object reached through `s` is `ResidentStudent`; after `s = nrs`, it becomes `NonResidentStudent`.

The call `s.getTuition()` compiles because `getTuition` is in `Student`'s expectation. The version that runs is selected at run time: after `s = rs`, the resident-student version runs; after `s = nrs`, the non-resident-student version runs. This is dynamic binding.

8. Static type for compilation, dynamic type for run time

Jackie summarized the recurring rule: compilation questions use static type; run-time tracing uses dynamic type. This applies not only to overridden method calls, but also to later topics such as casts and `ClassCastException`.

A key warning from the student questions was that `s.setPremiumRate()` does not compile merely because `s` currently points to a `ResidentStudent`. The compiler only sees the static type `Student`. Accessing child-specific behaviour through a parent-typed reference requires later tools such as casting, which Jackie postponed.

9. Multi-level hierarchies and cumulative expectation

The lecture then moved from one-level hierarchies to multi-level ones. The student hierarchy was extended conceptually with categories such as domestic, foreign, resident, and non-resident students. Jackie asked students to think about how Designs 1, 2, and 3 would scale in such a hierarchy: how many encodings, arrays, or parent-child classes would be needed?

The smartphone hierarchy then illustrated cumulative expectation. To determine the expectation of a class such as `IPhone13Pro`, `HuaweiP50Pro`, or `GalaxyS21Plus`, start at that class and collect methods upward through all ancestors.

10. Expectation versus `super`

A student asked why a class can have expectations inherited from a grandparent even though `super` only reaches the immediate parent. Jackie clarified that these are different concepts. Expectation is about the accumulated set of callable features from all ancestors. The keyword `super` is an implementation tool used inside a class to refer to the immediate parent's version of something.

Key Terms, Notation, and Structures

- **Static type**: the declared type of a reference variable; it determines compile-time expectations and does not change during execution.
- **Dynamic type**: the actual run-time class of the object currently reached by a reference variable; it may change when the variable is reassigned.

- **Expectation**: the set of attributes and methods that may legally be called on a reference variable; in this lecture, expectation is determined by static type.
- **Common expectation**: features declared in a parent class and inherited by subclasses; in the running example, these include the shared state and methods from `Student`.
- **Child-specific expectation**: features introduced only in a subclass; examples include `premiumRate` for residents and `discountRate` for non-residents.
- **Polymorphism**: the ability of a reference variable with a parent static type to point to an object of a child dynamic type.
- **Substitution / substitutability**: the rule that a child object may be used where a parent object is expected, but the reverse is not automatically safe.
- **Dynamic binding**: the run-time selection of the method version based on the dynamic type of the context object.
- **Overriding**: defining a child-class version of a method already available from a parent class, such as overriding `getTuition()` in `ResidentStudent`.
- **Valid upward assignment**: `s = rs` in the lecture's example, where a child reference is assigned to a parent variable.
- **Invalid downward assignment**: `rs = s` without an appropriate cast; accepting it would break the safety of resident-student expectations.
- **Private visibility**: `private` keeps an attribute or method visible only within the declaring class.
- **Protected visibility**: `protected` is useful for inheritance; Jackie framed it as a controlled way to let subclasses reuse selected parent attributes.
- `ClassCastException`: a run-time exception related to invalid casts; Jackie previewed it as a later inheritance topic.
- **Ancestor**: a parent, grandparent, or higher class in an inheritance hierarchy.
- **Smartphone hierarchy**: a multi-level hierarchy used to practise accumulating expectations, with methods such as `dial`, `surfWeb`, `FaceTime`, `Skype`, `QuickTake`, `ZoomImage`, and `SideSync`.

Key Examples and Demonstrations

Private first, protected when needed

Setup: A parent class `A` has a child class `B extends A`. Attributes in `A` may begin as `private`.

Main point: Start restrictive. If a subclass needs an inherited attribute to avoid duplicated declarations or duplicated logic, change that parent attribute to `protected`. This is a

design adjustment made during development, not something students must guess perfectly at the beginning.

What to practise: In Lab 4-style code, first implement cautiously, then inspect duplicated code. When a duplicated attribute clearly belongs in the parent, move it once and decide whether `protected` is justified.

Expectation table for `s`, `rs`, and `nrs`

Setup: `s` has static type `Student`; `rs` has static type `ResidentStudent`; `nrs` has static type `NonResidentStudent`.

Main point: `s` can access only the `Student` expectation. `rs` can access the inherited `Student` expectation plus resident-specific features. `nrs` can access the inherited `Student` expectation plus non-resident-specific features.

What to practise: Recreate the table from the notes. Mark each attempted call as compiling or not compiling, and justify each answer using only static type.

Why `rs = s` is invalid

Setup: `s` points to a plain `Student` object, and `rs` is a `ResidentStudent` variable.

Main point: If `rs = s` were allowed, `rs` could point to an object that does not contain resident-specific state such as `premiumRate`. Since `rs`'s static type promises resident-specific expectations, this would be unsafe.

What to practise: Reproduce Jackie's proof-by-contradiction in your own words: assume the assignment compiles, execute the arrow change, then explain which later access would reveal the contradiction.

Dynamic binding with `s.getTuition()`

Setup: `s` is declared as `Student`. First `s = rs`; later `s = nrs`. Both assignments are valid because the right-hand side types are lower in the hierarchy.

Main point: The call `s.getTuition()` is allowed because `getTuition` belongs to the static type `Student`. The version that runs depends on dynamic type: resident after `s = rs`; non-resident after `s = nrs`.

What to practise: Draw a four-row trace with columns for step, static type of `s`, dynamic type of `s`, and which `getTuition` version runs.

Smartphone expectations in a multi-level hierarchy

Setup: The smartphone hierarchy includes a `Smartphone` root, platform-level classes such as `iOS` and `Android`, and model-specific subclasses such as iPhone, Huawei, and Galaxy models.

Main point: Expectations accumulate from the class itself and all ancestors. The lecture derived model-specific sets: iPhone has `QuickTake` and `FaceTime` plus common phone features; Huawei has `ZoomImage` and `Skype` plus common phone features; Galaxy has `SideSync` and `Skype` plus common phone features.

What to practise: Pick another class in the hierarchy and list its expectation by walking upward to the root, removing duplicates when a method name appears at multiple levels.

Common Pitfalls and Misconceptions

Using dynamic type to answer a compilation question

Pitfall: Saying a call compiles because the object currently pointed to has the method.

Why it causes problems: The compiler checks static type only. For example, after `s = rs`, `s.setPremiumRate()` still does not compile if `s`'s static type is only `Student`.

How to avoid it: Before answering any "does it compile?" question, locate the variable declaration and use only the static type's expectation.

Assuming parent-to-child assignment is safe

Pitfall: Believing `rs = s` should compile just because the resident-student class is related to `Student`.

Why it causes problems: A parent object may lack child-specific fields and methods. A child-typed variable promises child expectations, so it cannot safely point to a plain parent object.

How to avoid it: Use the direction rule as an initial check: in a valid assignment without casting, the left static type should be higher up the hierarchy than the right static type.

Forgetting that static type does not change

Pitfall: Thinking that after `s = rs`, the variable `s` has somehow become a `ResidentStudent` variable.

Why it causes problems: The dynamic type of the object reached through `s` changes, but the static type of `s` remains `Student`. This affects which calls compile.

How to avoid it: In every trace, keep separate columns for static type and dynamic type. Only the dynamic type changes when the arrow changes.

Confusing expectation with `super`

Pitfall: Thinking that a class can only expect methods from its immediate parent because `super` only reaches the immediate parent.

Why it causes problems: Expectation accumulates from all ancestors, while `super` is only a mechanism used inside an implementation to call the immediate parent's version.

How to avoid it: Ask two separate questions: "What can I call from this static type?" for expectation, and "Which parent implementation am I reusing?" for `super`.

Making every inherited attribute `protected` immediately

Pitfall: Treating `protected` as the default choice just because a hierarchy exists.

Why it causes problems: It exposes more implementation detail than needed and weakens encapsulation.

How to avoid it: Follow Jackie's two-phase modifier habit: begin with `private`, then relax to `protected` only for attributes that subclasses genuinely need for shared implementation.

Delaying review of these foundations

Pitfall: Waiting until Written Test 2, Programming Test 3, or the final exam to review static type, dynamic type, expectation, and dynamic binding.

Why it causes problems: These concepts support later inheritance topics, including substitution rules, casting, `instanceof`, and `ClassCastException`.

How to avoid it: Review this lecture soon, retrace the diagrams, and ask questions before the next inheritance lecture builds on them.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain Jackie's two-phase rule for choosing between `private` and `protected`;
- define expectation as the set of callable attributes and methods for a reference variable;
- determine the expectation of a variable from its static type and its position in an inheritance hierarchy;
- distinguish common expectations inherited from a parent from child-specific expectations introduced in a subclass;
- decide whether calls such as `s.getTuition()`, `s.setPremiumRate()`, `rs.premiumRate`, and `nrs.discountRate` compile;
- explain why `s = rs` is valid but `rs = s` is not, using the hierarchy and expectation reasoning;
- trace a variable assignment by updating the reference arrow and the dynamic type while keeping the static type fixed;
- apply dynamic binding to predict which overridden method version runs after a polymorphic assignment;
- explain why `s.getTuition()` can compile using `Student`'s expectation but run a child-class version at run time;
- compare static-type questions with dynamic-type questions in a trace;
- compute the expectation of a class in a multi-level hierarchy by accumulating features from the class and all ancestors;

- distinguish cumulative expectation from the one-level behaviour of `super`;
- identify why `ClassCastException` is a natural next topic after static/dynamic type and polymorphism;
- act on the course logistics: submit Lab 3, start Lab 4 when released, prepare for Written Test 2, and note the tentative final-exam information.

Suggested Practice

- Redraw the `Student` hierarchy and make your own expectation table for `s`, `rs`, and `nrs`; mark each call as compile or no-compile before checking the notes.
- Recreate the argument for why `rs = s` is invalid. Do not only memorize the direction rule; explain the unsafe access that would occur if the compiler allowed it.
- Trace the dynamic-binding example twice: once after `s = rs` and once after `s = nrs`. For each call, write both why it compiles and which method version runs.
- Create two small examples of your own: one where static and dynamic types are the same, and one where a parent variable points to a child object.
- For the smartphone hierarchy, compute the expectations of at least three classes besides the three done in lecture.
- For Lab 4, start with `private` attributes where reasonable. When duplicated child code appears, decide carefully whether moving state to the parent and making it `protected` is justified.
- Review the upcoming Written Test 2 guide and example questions when Jackie posts them; use this lecture's examples as early practice for inheritance reasoning.
- Before the next lecture, review polymorphism and dynamic binding especially carefully, since Jackie said the next substitution rule depends on this foundation.

Where to Review

- **iPad/PDF notes, page 2**: assessment reminders, including Lab 3, Written Test 2, Programming Test 3, and the tentative final-exam information.
- **iPad/PDF notes, page 3**: the practical `private` versus `protected` rule for inherited attributes.
- **iPad/PDF notes, page 4**: definition of expectation and the table comparing `Student`, resident-student, and non-resident-student expectations.
- **iPad/PDF notes, page 5**: polymorphism intuition and why the assignment direction matters.

- **iPad/PDF notes, page 6**: dynamic-binding trace for `s.getTuition()` after assigning `s` to different child objects.
- **iPad/PDF notes, page 7**: the summary rule: compilation uses static type; run time uses dynamic type, with `ClassCastException` previewed.
- **iPad/PDF notes, pages 8–10**: multi-level student and smartphone hierarchies, including the expectation exercise for phone models.
- **Lecture recording**: revisit the segments on expectation, the invalid-assignment argument, dynamic binding, and the student question about expectation versus `super`.
- **Previous lecture materials**: review Lecture 16 for `extends`, `super`, child objects, and visibility modifiers if the vocabulary in this lecture feels shaky.
- **Labs and guides**: use Lab 4 for inheritance implementation practice and the Written Test 2 guide/example questions when they are released.

Closing Reminder

This digest is a roadmap for reviewing the lecture, not a substitute for the lecture itself. To learn this material, you need to revisit the diagrams, trace the assignments by hand, explain why calls compile or fail, and practise distinguishing static type from dynamic type. The next inheritance topics depend heavily on expectation, polymorphism, and dynamic binding, so use Jackie's notes, the recording, Lab 4, and the posted guides to check your understanding actively.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 18, July 9

Rule of Variable Substitutions, Dynamic Types, and Type Casting

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 11

Main Ideas

- The lecture formalized inheritance hierarchies by defining direct and indirect `extends` relationships, ancestors, and descendants.
- A class is counted as both its own ancestor and its own descendant, which matters when checking expectations and substitutions.
- Expectation still means the attributes and methods callable from a static type, and those expectations accumulate from ancestors to descendants.
- The main compile-time rule was the rule of substitution: in `oa = ob`, the static type of `ob` must be a descendant of the static type of `oa`.
- Compilation questions use static types; run-time tracing uses dynamic types.
- Polymorphism means one static type can have multiple possible dynamic types at run time, while dynamic binding chooses the method version from the current dynamic type.
- Type casting was motivated as a way to create a temporary alias with a more specific static type, so that child-specific expectations can be used when the programmer knows the dynamic type.
- Jackie used the compiler's inability to infer arbitrary dynamic behaviour, connected to undecidability and the Halting Problem, to explain why Java uses conservative static checks.
- Students were reminded that Lab 4 had been released, Written Test 2 was scheduled for July 20, Programming Test 3 for July 27, and the final exam was tentatively scheduled for August 10 from 9:00 AM to 12:00 noon.

Roadmap

1. Announcements and upcoming work

Jackie reminded students that the final exam was tentatively scheduled for Monday, August 10, from 9:00 AM to 12:00 noon, subject to final confirmation by the Registrar's Office. The immediate technical priority was the inheritance unit: Lab 4 had been released, Written Test 2 was scheduled for July 20, and Programming Test 3 for July 27.

The assessment connection for this lecture was explicit. Students should expect to draw or interpret inheritance hierarchies from `extends` declarations, apply compilation rules, and write explanations for why assignments compile or fail to compile.

2. Inheritance as a type hierarchy

The lecture began by formalizing diagrams such as `Z extends X` and `X extends Y`. The key idea is transitivity: if `Z` directly extends `X`, and `X` directly extends `Y`, then `Z` indirectly extends

`Y`. This is the same pattern students will use when reading larger hierarchies such as the smartphone hierarchy.

Jackie also reminded students that every class ultimately extends `Object`, even when that relationship is not written explicitly. This connects the inheritance discussion back to earlier work on `Object.equals`.

3. Ancestors, descendants, and expectation

For a fixed class `A`, the ancestors of `A` are `A` itself and all classes that `A` directly or indirectly extends. The descendants of `A` are `A` itself and all classes that directly or indirectly extend `A`. Classes outside both groups are neither ancestors nor descendants.

Expectation accumulates along the ancestor chain. If non-private members are declared in an ancestor, they are inherited by descendants. Therefore, the expectation of a descendant is at least as large as the expectation of its ancestor. There is no such guaranteed ordering for unrelated classes.

4. Smartphone hierarchy practice

The notes used the smartphone hierarchy to practise these definitions. For example, `GalaxyS21` has four ancestors when counted correctly: `GalaxyS21` itself, `Samsung`, `Android`, and `SmartPhone`. `SmartPhone` has only itself as an ancestor, but it has all classes in the hierarchy as descendants.

Jackie's warning was simple but important: do not forget the class itself when counting ancestors or descendants. This is a common off-by-one mistake in this topic.

5. From expectation to substitution

The lecture then moved from hierarchy vocabulary to the rule for variable assignment. For an assignment such as

```
oa = ob;
```

where `oa` has static type `A`, the question is what static type `ob` may have so that the assignment compiles.

The reasoning is expectation-based. After the assignment, any use of `oa.` still expects the features promised by static type `A`. Therefore the object reached through `ob` must be able to fulfill at least those expectations. This is guaranteed when the static type of `ob` is a descendant of `A`.

6. The rule of substitution

The rule can be summarized as follows: in an assignment `var1 = var2`, the assignment compiles only if the static type of `var2` is a descendant of the static type of `var1`. Equivalently, the expectation of the right-hand static type must be at least as large as the expectation of the left-hand static type.

Jackie emphasized that this is a compilation rule, so students should look only at static types. Dynamic types matter later, after the code has already compiled and is being traced at run time.

7. Applying the substitution rule

The lecture applied the rule to several smartphone and student examples. Assigning an `IPhoneSE` or `IPhone13Pro` reference to an `iOS` variable compiles because those right-hand static types are descendants of `iOS`. Assigning a `SmartPhone` reference to an `iOS` variable fails because `SmartPhone` is an ancestor, not a descendant, of `iOS`. Assigning a `HuaweiP50Pro` reference to an `iOS` variable also fails because that class is neither an `iOS` descendant nor able to fulfill the `iOS` expectation.

The same rule explains why `Student jim = new ResidentStudent(...)` compiles, while `ResidentStudent jeremy = new Student(...)` does not. A plain `Student` cannot fulfill resident-specific expectations such as premium-rate features.

8. Change of dynamic type

Once assignments pass the static-type rule, they can change dynamic type at run time. In the student trace, a variable such as `jim` may have static type `Student` throughout the fragment, but its dynamic type may change from `Student` to `ResidentStudent`, and later to `NonResidentStudent`, depending on assignments such as `jim = rs` and `jim = nrs`.

This is why a trace should keep static type and dynamic type separate. Static type explains whether the line compiles; dynamic type explains which object is reached after the assignment and which overridden method body runs.

9. Polymorphism and dynamic binding

Jackie then named the two ideas behind the traces. Polymorphism means that a variable of one static type may have multiple possible dynamic types at run time. For example, a `Student` variable may point to a `ResidentStudent` object at one time and a `NonResidentStudent` object later.

Dynamic binding is the consequence for overridden methods: the static type says the call is legal, while the current dynamic type selects the version that executes.

The smartphone example repeated the same idea with `surfWeb`: before reassignment the call followed the iOS side; after `sp1 = sp2`, it followed the Android side.

10. Why the compiler cannot simply infer the dynamic type

To motivate casting, Jackie asked why the compiler cannot just notice that a parent-typed variable currently points to a child object. The answer is not just that the compiler is inconvenient; inferring the dynamic behaviour of an arbitrary program is undecidable in general. If a program contains loops that might not terminate, a tool trying to simulate the program to discover the final dynamic type might also never terminate.

The A-plus challenge was presented as a thought experiment: write a program that reads another Java program and a variable name, then reports that variable's final dynamic type. Jackie immediately explained that solving this generally would deserve far more than a course grade, because it is connected to the Halting Problem. The practical takeaway is that Java uses static, conservative rules instead of attempting impossible general run-time inference.

11. Type cast motivation and syntax

The lecture ended by motivating type casting. Suppose `jim` has static type `Student` but currently points to a `ResidentStudent` object. The assignment

```
ResidentStudent rs = jim;
```

does not compile because the static type `Student` is not a descendant of `ResidentStudent`. But if the programmer knows that `jim`'s dynamic type is currently `ResidentStudent`, a cast can create a temporary expression with static type `ResidentStudent`:

```
ResidentStudent rs = (ResidentStudent) jim;
```

This does not change `jim`'s static type. It creates a cast expression, or alias, whose static type is the cast type. That alias can then be assigned to a `ResidentStudent` variable, allowing resident-specific calls such as `rs.setPremiumRate(1.5)`. Jackie also warned that casting must be handled carefully because a cast that compiles can still fail at run time with a `ClassCastException`.

Key Terms, Notation, and Structures

- **Direct** `extends`: A relationship written explicitly in Java, such as `Z extends X`.
- **Indirect** `extends`: A transitive relationship through one or more intermediate classes; if `Z extends X` and `X extends Y`, then `Z` indirectly extends `Y`.
- **Ancestor of** `A`: `A` itself, together with every class that `A` directly or indirectly extends.
- **Descendant of** `A`: `A` itself, together with every class that directly or indirectly extends `A`.
- **Expectation**: The set of attributes and methods that may legally be called from a reference variable's static type.
- **Expectation ordering**: Along an inheritance chain, descendants have expectations at least as large as their ancestors, assuming the inherited members are non-private.
- **Rule of substitution**: In `var1 = var2`, the assignment compiles only if the static type of `var2` is a descendant of the static type of `var1`.
- **Static type**: The declared type of a variable; it is fixed after declaration and controls compilation.

- **Dynamic type**: The actual run-time class of the object currently reached through a reference; it can change when the reference is reassigned.
- **Polymorphism**: The situation where one static type may allow multiple possible dynamic types at run time.
- **Dynamic binding**: The rule that the version of an overridden method being invoked is determined by the current dynamic type.
- **Type cast expression**: An expression such as `(ResidentStudent) jim` that asks the compiler to treat the expression as having the cast type.
- **Cast type**: The type written in parentheses in a cast expression; it becomes the static type of that cast expression, not of the original variable.
- **Alias**: Jackie's way of describing the temporary reference produced by a cast expression; it refers to the same object but has a different static type for that expression.
- `ClassCastException`: The run-time exception Jackie warned can occur when a cast compiles but the actual object is not compatible with the cast type.
- **Undecidable problem**: A problem for which no algorithm can correctly solve all possible inputs; Jackie used the final-dynamic-type challenge to connect compiler limitations to the Halting Problem.
- **Main hierarchies**: The lecture used a smartphone hierarchy and a student hierarchy involving `Student`, `ResidentStudent`, and `NonResidentStudent`.

Key Examples and Demonstrations

Transitivity and ancestors

Setup: The notes used a small chain such as `Z extends X` and `X extends Y`, then connected it to larger hierarchies such as smartphones.

Main point: Direct `extends` arrows create indirect relationships through transitivity. A class is also counted as its own ancestor and descendant.

What to practise: Given any hierarchy, pick one class, color or list its ancestors, color or list its descendants, and identify the classes that are neither.

Galaxy S21 ancestor count

Setup: In the smartphone hierarchy, `GalaxyS21` is under `Samsung`, which is under `Android`, which is under `SmartPhone`.

Main point: The ancestors are `GalaxyS21`, `Samsung`, `Android`, and `SmartPhone`. The count is four because the class itself must be included.

What to practise: Redo this count for `SmartPhone`, `Android`, `iOS`, and one iPhone class, checking whether each answer includes the class itself.

Deriving the rule of substitution

Setup: The notes fixed a variable `oa` with static type `A` and asked what static type `ob` may have in `oa = ob`.

Main point: After substitution, `oa` still has `A`'s expectation. Therefore `ob`'s static type must be able to fulfill `A`'s expectation, which is guaranteed when `ob`'s static type is a descendant of `A`.

What to practise: Write both explanations for the same assignment: one using "descendant of," and one using "fulfills the expectation of."

Smartphone substitution checks

Setup: The notes used variables such as `iOS sp1`, `IPhoneSE sp2`, `IPhone13Pro sp3`, `SmartPhone sp2`, and `HuaweiP50Pro sp2`.

Main point: Assignments from iPhone-specific variables into an `iOS` variable compile because iPhone classes are descendants of `iOS`. Assigning a `SmartPhone` or `HuaweiP50Pro` variable into an `iOS` variable fails because those static types cannot fulfill the `iOS` expectation.

What to practise: For each assignment, locate the left static type and right static type in the diagram before answering. Do not use intuition about object names alone.

Student dynamic type trace

Setup: A fragment declares `Student jim`, `ResidentStudent rs`, and `NonResidentStudent nrs`; then it executes assignments such as `jim = rs` and `jim = nrs`.

Main point: The assignments compile because the right-hand static types are descendants of `Student`. At run time, `jim`'s dynamic type changes to the dynamic type of the object reached by the right-hand side.

What to practise: Make a trace table with columns for line number, whether the line compiles, `jim`'s static type, `jim`'s dynamic type, and which `getTuition` version would run.

Polymorphism and `surfWeb`

Setup: `SmartPhone sp1` initially points to an `IPhone13Pro`, while `SmartPhone sp2` points to a `GalaxyS21`. The method `surfWeb` appears along the hierarchy.

Main point: The call `sp1.surfWeb()` is allowed because `surfWeb` is in the static expectation of `SmartPhone`. The version that runs depends on dynamic type: before reassignment it follows the iOS/iPhone side; after `sp1 = sp2`, it follows the Android/Galaxy side.

What to practise: Redraw the arrows before and after `sp1 = sp2`, then explain which `surfWeb` version is selected and why.

Type casting to access `setPremiumRate`

Setup: `Student jim = new ResidentStudent(...)` gives `jim` static type `Student` but dynamic type `ResidentStudent`. The direct assignment `ResidentStudent rs = jim` fails to compile.

Main point: The cast `(ResidentStudent) jim` creates a cast expression with static type `ResidentStudent`. This can be assigned to a `ResidentStudent` variable, after which `rs.setPremiumRate(1.5)` is within the static expectation.

What to practise: Explain why `jim.setPremiumRate(1.5)` still does not compile, even though `jim` currently points to a `ResidentStudent` object.

The A-plus challenge as a compiler-limits example

Setup: Jackie described a hypothetical program that would take another Java program and a variable name, then output the final dynamic type of that variable.

Main point: The general problem is undecidable. A program being analyzed may contain loops or other behaviours that prevent any complete simulation from finishing.

What to practise: Use this example to explain why compilers rely on static types instead of trying to infer all run-time behaviour.

Common Pitfalls and Misconceptions

Forgetting to include the class itself

Pitfall: Counting only the classes above or below a class when listing ancestors or descendants.

Why it causes problems: The formal definitions include the class itself. Forgetting this gives wrong counts, such as saying `GalaxyS21` has only three ancestors.

How to avoid it: Start every ancestor or descendant list with the class itself, then move upward or downward through the hierarchy.

Reversing the substitution rule

Pitfall: Treating any assignment between related types as valid, or allowing ancestor-to-descendant assignment without a cast.

Why it causes problems: A parent or ancestor object may not satisfy child-specific expectations. For example, a plain `Student` cannot fulfill `ResidentStudent` expectations.

How to avoid it: For `left = right`, check whether the right static type is a descendant of the left static type.

Using dynamic type to answer "does it compile?"

Pitfall: Saying a line compiles because, at that moment, a variable is pointing to an object of a useful dynamic type.

Why it causes problems: The compiler does not trace run-time arrows. It checks only static types and their expectations.

How to avoid it: When the question is compilation, ignore the current object diagram except for locating declared static types.

Mixing up polymorphism and dynamic binding

Pitfall: Treating polymorphism and dynamic binding as the same rule.

Why it causes problems: Polymorphism is about one static type having many possible dynamic types. Dynamic binding is the separate run-time rule that chooses which overridden method version to execute.

How to avoid it: First ask what dynamic types the static type may allow; then ask which method version the current dynamic type selects.

Assuming casting changes the original variable's static type

Pitfall: Thinking that after writing `(ResidentStudent) jim`, the variable `jim` has permanently become a `ResidentStudent` variable.

Why it causes problems: Static types never change after declaration. A cast expression creates a temporary alias with the cast type; it does not rewrite the declaration of `jim`.

How to avoid it: Treat `(Type) var` as a new expression on the right-hand side, not as a mutation of `var`.

Believing every compiling cast is safe

Pitfall: Using casts merely to silence the compiler.

Why it causes problems: Jackie warned that casting has both a compile-time dimension and a run-time dimension. A cast may compile but still fail at run time with `ClassCastException` if the dynamic type is not compatible.

How to avoid it: Before casting, be able to explain what the current dynamic type is and why it is compatible with the cast type.

Skipping written explanations

Pitfall: Only marking assignments as "compile" or "fail" without being able to justify the decision.

Why it causes problems: Jackie explicitly connected this skill to Written Test 2 and exam-style questions.

How to avoid it: Practise full sentences: "This compiles because the static type of the right-hand side is a descendant of the static type of the left-hand side."

What You Should Be Able to Do

After this lecture, you should be able to:

- draw an inheritance hierarchy from `extends` declarations, including direct and indirect relationships;
- identify ancestors, descendants, and neither-ancestor-nor-descendant classes for a chosen class;

- remember that a class is included in both its own ancestor set and its own descendant set;
- explain how expectation accumulates from ancestors to descendants;
- apply the rule of substitution to decide whether an assignment compiles;
- justify a substitution decision using both the descendant explanation and the expectation explanation;
- distinguish static type questions from dynamic type questions;
- trace how a variable's dynamic type changes after valid assignments;
- use dynamic binding to predict which overridden method version runs;
- explain polymorphism as one static type having multiple possible dynamic types;
- explain why compilers use conservative static checks rather than trying to infer arbitrary run-time behaviour;
- describe what a type cast expression does and what it does not do;
- explain why `ResidentStudent rs = jim` may fail to compile while `ResidentStudent rs = (ResidentStudent) jim` can compile;
- state why careless casting can lead to `ClassCastException`;
- prepare for Lab 4, Written Test 2, and Programming Test 3 by practising hierarchy diagrams, substitution checks, traces, and cast reasoning.

Suggested Practice

- Redraw the smartphone hierarchy and, for at least five classes, list ancestors, descendants, and unrelated classes.
- For each class you choose, write its expectation by walking up the hierarchy and collecting inherited methods.
- Use the posted template to practise substitution checks. For each assignment, write a one-sentence justification, not just a yes/no answer.
- Rework the student examples: decide whether each assignment compiles, then trace the dynamic type of `jim`, `rs`, or `nrs` after each line.
- Rework the smartphone `surfWeb` trace and explain when the iOS-side version versus Android-side version is invoked.
- Write a small Java fragment with `Student`, `ResidentStudent`, and `NonResidentStudent`; test which assignments compile before and after adding an explicit cast.

- When experimenting with casts, intentionally try one cast that should be unsafe and observe the `ClassCastException`; then explain why it occurred.
- Start Lab 4 early, since it is the main current practice vehicle for inheritance hierarchy design.
- Before Written Test 2, practise explaining compile errors in words using "static type," "descendant," and "expectation."

Where to Review

- **iPad/PDF notes, pages 1–2**: lecture title and announcements, including Lab 4, Written Test 2, Programming Test 3, and the tentative final exam time.
- **iPad/PDF notes, pages 3–4**: inheritance as a type hierarchy, ancestors, descendants, and expectation accumulation.
- **iPad/PDF notes, page 5**: smartphone variables `sp1`, `sp2`, and `sp3`; practise what can appear on the right-hand side of each assignment.
- **iPad/PDF notes, pages 6–7**: the rule of substitution and the expectation-based derivation using `oa = ob`.
- **iPad/PDF notes, pages 8–10**: substitution examples using the smartphone hierarchy.
- **iPad/PDF notes, pages 11–17**: static type versus dynamic type, valid and invalid student substitutions, and dynamic-type changes.
- **iPad/PDF notes, page 18**: definitions and examples for polymorphism and dynamic binding.
- **iPad/PDF notes, pages 19–21**: type cast motivation, the A-plus challenge, and the anatomy of a cast expression.
- **Lecture recording**: revisit Jackie's explanations of why the substitution rule is based on static types, why the compiler cannot infer arbitrary dynamic behaviour, and how the cast expression creates a temporary alias.
- **Lab 4 and Written Test 2 preparation materials**: use them to practise hierarchy diagrams, assignment rules, and written justifications.

Closing Reminder

This digest is only a roadmap. The main learning work is to revisit Jackie's iPad/PDF notes and the lecture recording, redraw the hierarchies, check assignments using static types, trace dynamic types at run time, and practise explaining your answers in words. The substitution rule, polymorphism, dynamic binding, and casting syntax will keep reappearing in inheritance labs, Written Test 2, Programming Test 3, and the final exam.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 19, July 14

Type Casts: Syntax, Compilation Rules, and Runtime Safety

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 10

Closing Reminder 11

Main Ideas

- The lecture continued the inheritance unit by focusing on type casts as a controlled way to create a new static view of an existing object.
- Jackie reviewed the rule of substitution: in an assignment, the right-hand side static type must be a descendant of the left-hand side static type.
- A method call such as `aPhone.FaceTime()` is checked against the static type of `aPhone`, even if the current dynamic object supports the method.
- A type cast creates an alias with the cast type as its static type; it does not change the original variable's static type or the object itself.
- The lecture distinguished named casts, where the cast result is stored in a variable, from anonymous casts, where the cast result is used directly.
- Anonymous casts require careful parentheses because the dot operator has higher precedence than the cast operator.
- For compilation, a cast may go upward to an ancestor or downward to a descendant of the variable's static type; the dynamic type is ignored at this stage.
- For run time, a compilable cast can still fail with `ClassCastException` when the dynamic type cannot fulfill the expectation of the cast type.
- Jackie emphasized that this compilation-versus-run-time analysis is part of the coverage for Written Test 2 and will also matter for the final exam.

Roadmap

1. Announcements and assessment context

Jackie confirmed that the final exam had been finalized for Monday, August 10, from 9:00 AM to 12:00 noon, with the room still to be announced. The format will use question and answer booklets rather than Scantrons only, though multiple-choice questions may still appear. Students registered for university accommodations must arrange their exam with the accessibility office on time.

For the near term, Lab 4 had been released and was identified as the best preparation for Programming Test 3. Written Test 2 was scheduled for July 20, with the guide and example questions released. Jackie also noted that the material in this lecture completes most of the scope for Written Test 2, except for the short continuation of the final chained-cast example at the beginning of the next lecture.

2. Review: the rule of substitution

The technical work began by revisiting the rule of substitution from the previous inheritance lecture. For an assignment, compare the static type on the left with the static type on the right. The right-hand side static type must be a descendant of the left-hand side static type.

Jackie emphasized that this rule is best understood through expectation. The assigned object must be able to fulfill everything expected by the left-hand side static type. This rule underlies both ordinary polymorphic assignments and the cast-assignment examples later in the lecture.

3. Why type casting is needed

The lecture returned to the problem that motivates casting. A variable may have a broad static type but a more specific dynamic type. For example, `SmartPhone aPhone = new IPhone13Pro();` is valid because `IPhone13Pro` is a descendant of `SmartPhone`. However, `aPhone.FaceTime()` does not compile because the compiler checks only the static type `SmartPhone`, and `FaceTime` is not part of that expectation.

This is the gap that type casting addresses. If the programmer knows the current object has a more specific type, a cast can create a new alias whose static type exposes the needed method.

4. The four types involved in a cast assignment

Jackie warned that several types may appear in one cast expression. Students should identify them explicitly instead of relying on intuition.

In an expression such as

```
iOS myIPhone = (IPhone13Pro) aPhone;
```

the original variable `aPhone` has its own static type and dynamic type, the expression has a cast type `IPhone13Pro`, and the left-hand side variable `myIPhone` has static type `iOS`. The assignment compiles because the cast expression has static type `IPhone13Pro`, which is a descendant of `iOS`.

5. Named casts and anonymous casts

A named cast stores the cast result in a variable, such as assigning `(IPhone13Pro) aPhone` to `myIPhone`. This is useful when the same casted view will be used more than once, because it gives the alias a name.

An anonymous cast uses the cast expression directly, such as:

```
((IPhone13Pro) aPhone).FaceTime();
```

This is concise when the casted view is only needed for one call. The cast expression has static type `IPhone13Pro`, so the method call is checked against the expectation of `IPhone13Pro`.

6. Parentheses and operator precedence

Jackie highlighted a syntax trap. The expression

```
(IPhone13Pro) aPhone.FaceTime();
```

is not the same as the correct anonymous cast. Java gives the dot operator higher precedence than the cast operator, so the compiler tries to check `aPhone.FaceTime()` first. That fails because `aPhone` has static type `SmartPhone`. To force the cast to happen first, write the extra parentheses around the whole cast expression.

7. Compilable casts: upward versus downward

The next part focused only on compilation. When checking whether a single cast compiles, ignore the dynamic type. Look only at the static type of the expression being cast.

If `Android myPhone = new GalaxyS21Plus();`, then casts from `myPhone` to ancestors of `Android` are upward casts, and casts from `myPhone` to descendants of `Android` are downward casts. Both kinds can compile. Upward casting reduces expectation, such as viewing an `Android` reference as a `SmartPhone`. Downward casting increases expectation, such as viewing it as a `GalaxyS21Plus`.

8. Runtime casts and ClassCastException

After establishing which casts compile, Jackie brought dynamic type back into the analysis. A cast that compiles can still fail at run time if the actual dynamic type cannot fulfill the expectation of the cast type.

The student example showed this clearly:

```
Student jim = new NonResidentStudent(...);
ResidentStudent rs = (ResidentStudent) jim;
rs.setPremiumRate(1.5);
```

All three lines pass compilation checks. At run time, the cast on the second line throws `ClassCastException`. The dynamic type of `jim` is `NonResidentStudent`, which is not a descendant of the cast type `ResidentStudent`. The exception occurs at the cast, before the later method call is attempted.

9. Smartphone examples for runtime safety

The smartphone hierarchy repeated the same run-time analysis. If `aPhone` has static type `SmartPhone` but dynamic type `GalaxyS21Plus`, then a cast to `IPhone13Pro` compiles as a downward cast from `SmartPhone`. It still fails at run time because `GalaxyS21Plus` cannot fulfill the expectation of `IPhone13Pro`.

A student suggested changing the cast target to `Android`. Jackie used this to show the positive case: `GalaxyS21Plus` is a descendant of `Android`, so the cast to `Android` can be exception-free, provided the rest of the assignment and method calls are also type-correct.

10. Compilable versus exception-free casts

The summary diagram fixed `Android` as the static type and `Samsung` as the dynamic type. Relative to the static type, casts to ancestors or descendants of `Android` are compilable, while casts to unrelated branches such as the `iOS` side are non-compilable.

Among the compilable casts, only those whose cast type can be fulfilled by the dynamic type are exception-free. If the dynamic type is `Samsung`, then casts to `Samsung` and its ancestors are safe. Casts to other Android-side descendants, such as a Huawei branch or a Samsung child class not actually present as the dynamic type, can compile but throw `ClassCastException`.

11. Chained casts and the cliffhanger for next lecture

The final exercise used symbolic classes `A`, `B extends A`, `C extends B`, and `D extends A`. Jackie first had students draw the hierarchy and check the direct cast `(D) b`. Since `D` is neither an ancestor nor a descendant of `B`, the direct cast does not compile.

Then Jackie showed how to make the compiler accept the idea by chaining casts:

```
D d = (D) (A) b;
```

The first cast moves upward from `B` to `A`; the second moves downward from `A` to `D`. Jackie noted that this generalizes through `Object`: `(T2)(Object)obj` can make very broad cast patterns compile. The unresolved question was whether this avoids `ClassCastException`; students were asked to think about the dynamic type and this example was left for the next lecture.

Key Terms, Notation, and Structures

- **Rule of substitution**: In an assignment, the right-hand side static type must be a descendant of the left-hand side static type.
- **Static type**: The declared type of a variable or expression; it controls compilation and cannot be changed after declaration.
- **Dynamic type**: The actual run-time type of the object currently referenced; it can change when a reference is reassigned.
- **Expectation**: The set of attributes and methods that may be called through a static type.
- **Type cast**: A Java expression such as `(IPhone13Pro) aPhone` that creates a new static view of the same object.
- **Cast type**: The type written inside the cast parentheses; it becomes the static type of the cast expression.
- **Cast expression**: The full expression produced by a cast, such as `(ResidentStudent) jim`.

- **Alias**: A reference to the same object with a possibly different static type; casting creates an alias, not a new object and not a new static type for the original variable.
- **Named cast**: A cast whose result is stored in a variable, such as `iOS myIPhone = (IPhone13Pro) aPhone;`.
- **Anonymous cast**: A cast used directly without a new variable, such as `((IPhone13Pro) aPhone).FaceTime();`.
- **Operator precedence**: The rule that `.` binds before a cast; this is why anonymous casts need the extra parentheses.
- **Upward cast**: A cast from a static type to one of its ancestors; it reduces expectation.
- **Downward cast**: A cast from a static type to one of its descendants; it increases expectation and can be risky at run time.
- **Compilable cast**: A cast whose cast type is either an ancestor or a descendant of the expression's static type.
- **Exception-free cast**: A compilable cast whose cast type can be fulfilled by the current dynamic type.
- **ClassCastException**: A run-time exception thrown at the cast expression when the dynamic type is incompatible with the cast type.
- **Chained cast**: A sequence such as `(D)(A)b` or `(T2)(Object)obj`, where each cast is checked separately.

Key Examples and Demonstrations

1. Calling FaceTime through a SmartPhone reference

Setup: `SmartPhone aPhone = new IPhone13Pro();` gives `aPhone` static type `SmartPhone` and dynamic type `IPhone13Pro`.

Main point: `aPhone.FaceTime()` does not compile because `FaceTime` is not in the expectation of `SmartPhone`. The dynamic type does not rescue the call at compile time.

What to practise: For every method call, write down the context object's static type first, then decide whether the method is expected from that static type.

2. Named and anonymous casts

Setup: The lecture used `iOS myIPhone = (IPhone13Pro) aPhone;` and `((IPhone13Pro) aPhone).FaceTime();`.

Main point: Both create a cast expression with static type `IPhone13Pro`. The named version stores the alias in a variable; the anonymous version uses the alias immediately.

What to practise: Write both versions in Eclipse or on paper, and explain which static type is used to check the method call.

3. Upward versus downward casts

Setup: In `Android myPhone = new GalaxyS21Plus();`, Jackie compared `(SmartPhone) myPhone` with `(GalaxyS21Plus) myPhone`.

Main point: The upward cast to `SmartPhone` compiles and reduces the available expectation. The downward cast to `GalaxyS21Plus` compiles and increases the available expectation, but must later be checked at run time.

What to practise: Redraw the smartphone hierarchy, mark the static type `Android`, and list which casts are upward, downward, or neither.

4. ClassCastException in the student hierarchy

Setup: `Student jim = new NonResidentStudent(...);` followed by `ResidentStudent rs = (ResidentStudent) jim;`.

Main point: The cast compiles because `ResidentStudent` is a descendant of the static type `Student`. It fails at run time because the dynamic type `NonResidentStudent` is not a descendant of the cast type `ResidentStudent`.

What to practise: Identify the exact line where the exception occurs. The exception is thrown at the cast expression, not at the later call to `setPremiumRate`.

5. Compilable, non-compilable, exception-free, and exception-throwing regions

Setup: The summary diagram used `Android myPhone = new Samsung();` and classified possible cast targets in the smartphone hierarchy.

Main point: Static type determines which casts compile; dynamic type determines which of the compilable casts are exception-free. These are two separate filters.

What to practise: Pick another static/dynamic pair in the hierarchy and create the same four-region classification yourself.

6. Chained casts through an ancestor

Setup: With `B extends A`, `C extends B`, and `D extends A`, a variable `b` has static type `B`. A direct cast to `D` is neither upward nor downward.

Main point: `(D)(A)b` compiles because it splits the sideways cast into an upward cast followed by a downward cast. This is a compile-time trick, not proof of run-time safety.

What to practise: Trace the dynamic type of `b` and decide whether the chained cast should be exception-free. This is exactly the question Jackie left for the next lecture.

Common Pitfalls and Misconceptions

1. Using dynamic type to justify a method call at compile time

Pitfall: Saying that `aPhone.FaceTime()` compiles because `aPhone` currently points to an `IPhone13Pro` object.

Why it causes problems: The compiler checks the static type `SmartPhone`, not the dynamic type. If `FaceTime` is not expected from `SmartPhone`, the call fails to compile.

How to avoid it: For compilation questions, ignore the dynamic type until after you have checked the static type.

2. Thinking a cast changes the original variable's static type

Pitfall: Treating `(ResidentStudent) jim` as if it permanently changes `jim` from static type `Student` to static type `ResidentStudent`.

Why it causes problems: Static type is fixed at declaration. The cast expression only creates a new alias with the cast type as its static type.

How to avoid it: After every cast, ask which expression or variable has the new static type. Do not assign that change to the original variable unless a new variable is actually declared.

3. Forgetting the extra parentheses in anonymous casts

Pitfall: Writing `(IPhone13Pro) aPhone.FaceTime()` when you mean to cast `aPhone` before calling `FaceTime`.

Why it causes problems: The dot operator is evaluated before the cast, so Java checks `aPhone.FaceTime()` first and the code fails.

How to avoid it: Use `((IPhone13Pro) aPhone).FaceTime();` and be able to explain why the outer parentheses are needed.

4. Assuming every compilable cast is safe

Pitfall: Stopping the analysis once a downward cast compiles.

Why it causes problems: Compilation only checks the relationship between the cast type and the source static type. A downward cast can still throw `ClassCastException` when executed.

How to avoid it: Use a two-stage check: first static type for compilation, then dynamic type for run-time safety.

5. Placing ClassCastException on the later method call

Pitfall: Saying the exception occurs when `rs.setPremiumRate(1.5)` tries to use a resident-specific feature.

Why it causes problems: Java detects the invalid cast immediately. The program does not wait until the later method call.

How to avoid it: Mark the cast expression itself as an executable run-time check.

6. Treating sibling classes as interchangeable because they look similar

Pitfall: Reasoning that a cast between `ResidentStudent` and `NonResidentStudent` should work if they happen to have similar methods.

Why it causes problems: Java casting uses the declared inheritance hierarchy, not structural similarity between fields and methods.

How to avoid it: Draw the hierarchy and ask only whether the dynamic type is a descendant of the cast type.

7. Believing a chained cast guarantees run-time success

Pitfall: Assuming `(D)(A)b` is safe just because both individual casts compile.

Why it causes problems: Chaining can satisfy the compiler while still producing a cast type that the dynamic object cannot fulfill.

How to avoid it: After any chained cast, redo the dynamic-type check for the final cast type.

What You Should Be Able to Do

After this lecture, you should be able to:

- Apply the rule of substitution to ordinary assignments and cast assignments.
- Explain why `var.method()` is checked against `var`'s static type, not its current dynamic type.
- Identify the static type, dynamic type, cast type, and destination static type in a cast-assignment example.
- Explain why type casting creates an alias rather than changing the original variable or allocating a new object.
- Write both named and anonymous cast expressions correctly.
- Justify why `((Type) var).method()` needs the extra parentheses.
- Distinguish upward casts from downward casts using an inheritance hierarchy.
- Predict whether a cast compiles by comparing the cast type with the expression's static type.
- Predict whether a compilable cast is exception-free by comparing the dynamic type with the cast type.
- Locate the exact line that throws `ClassCastException` in a short code fragment.
- Classify possible cast targets as non-compilable, compilable but exception-throwing, or exception-free.
- Explain why sibling classes cannot be cast directly unless the cast is routed through an ancestor or `Object`.
- Draw an inheritance diagram from `extends` declarations before analyzing casts.
- Prepare for Written Test 2 by writing complete explanations, not just yes/no answers.

Suggested Practice

- Rewatch or review the previous asynchronous lecture's rule of substitution before practising this lecture's casting examples.
- Redraw the smartphone hierarchy and mark the static type, dynamic type, and cast type for each example on pages 3, 5, 6, 7, 8, 11, and 12 of Jackie's iPad notes.
- For each cast example, write two sentences: one explaining compilation using static types, and one explaining run-time safety using dynamic type.
- Practise the named cast and anonymous cast syntax in a small Java file, including the incorrect anonymous form, so that the precedence issue is visible.
- Use the student hierarchy to create one cast that compiles and succeeds, one cast that compiles and throws `ClassCastException`, and one cast that fails to compile.
- Redo the blank in-class exercises from the slides or notes and write explanations in the style expected on Written Test 2.
- Work actively on Lab 4, because Jackie identified it as the best preparation for Programming Test 3.
- Read the supplemental PDF on static type, expectation, dynamic types, and type casting under the inheritance required readings.
- After practising chained casts, ask whether the final dynamic type can fulfill the final cast type instead of stopping at compilation.

Where to Review

- **iPad notes page 1**: Lecture title and the three main technical targets: named versus anonymous casts, compilable casts, and `ClassCastException`.
- **iPad notes page 2**: Final exam, Written Test 2, Programming Test 3, and Lab 4 reminders.
- **iPad notes page 3**: Named versus anonymous casts using the smartphone hierarchy and the `aPhone.FaceTime()` example.
- **iPad notes page 4**: Slides 60–66 and the link to the required reading on type casting.
- **iPad notes page 5**: Upward and downward casts from `Android myPhone = new GalaxyS21Plus();`, including reduced versus increased expectation.
- **iPad notes pages 6–8**: Examples of compilable casts that may throw exceptions, plus the `Android` variant.

- **iPad notes pages 9–10**: Additional practice questions on whether a cast compiles and whether it fails at run time.
- **iPad notes page 11**: Summary diagram separating compilable casts, non-compilable casts, exception-free casts, and `ClassCastException` cases.
- **iPad notes page 12**: Symbolic `A/B/C/D` hierarchy and the chained-cast idea `(D)(A)b`.
- **Lecture recording**: Review the rule-of-substitution recap, the operator-precedence warning, and the two-stage cast analysis.
- **Required reading**: Use the posted PDF on static type, expectation, dynamic types, and type casting as a compact supplement to the lecture.
- **Assessment materials**: Use the Written Test 2 guide, example questions, and practice test to practise complete written explanations.

Closing Reminder

This digest is only a roadmap for Lecture 19. The key work is to return to Jackie's iPad notes, the lecture recording, the required reading, and the practice questions, then actively trace casts yourself. For each example, separate the static-type compilation check from the dynamic-type run-time check. Reading the digest alone is not enough preparation for Written Test 2, Programming Test 3, or the final exam.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 20, July 16

`instanceof` Guards, Safe Casts, and Call-by-Value Substitution

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 6

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 11

Main Ideas

- The lecture finished the chained-cast example from the previous class, showing that a cast can be made compilable while still failing at run time.
- Jackie introduced `instanceof` as a Boolean operator for checking dynamic type compatibility before performing a risky cast.
- The compile-time rule for `x instanceof Y` is tied to the corresponding cast: it compiles exactly when `(Y) x` would compile.
- The run-time rule for `instanceof` depends on dynamic type: `x instanceof Y` is true when the dynamic type of `x` can fulfill the expectation of `Y`.
- A true `instanceof` check can be used as a guard for a subsequent cast, provided the dynamic type of the reference has not changed.
- The lecture contrasted `instanceof`, which evaluates to true or false, with casts, which either create an alias or throw `ClassCastException`.
- The smartphone examples reinforced that method calls compile according to static type, not dynamic type, even when the dynamic type has more methods.
- The lecture closed by reconnecting call by value with inheritance: parameter passing behaves like an implicit assignment and must satisfy the rule of substitution.
- Jackie emphasized that the first chained-cast exercise completed the Written Test 2 scope, while the new `instanceof` and call-by-value material is for later study and the final exam.

Roadmap

1. Announcements and assessment context

Jackie repeated the main course reminders. The final exam had been finalized for Monday, August 10, from 9:00 AM to 12:00 noon, with the room still to be announced. Written Test 2 was scheduled for Monday, July 20, during the regular lab session; students were told to use the guide and example questions.

The first few minutes finishing the chained-cast exercise remained part of Written Test 2 coverage. Jackie explicitly said that the later `instanceof` material and the brief call-by-value connection would not be on Written Test 2, but would matter for the final exam. Programming Test 3 was scheduled for the week of July 27, with guide and example questions expected before students needed to begin focused preparation. Programming Test 2 had finished automated grading, with some manual grading still pending.

2. Finishing the chained-cast exercise

The technical work began by returning to the symbolic hierarchy from the previous lecture:

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

with `B b = new C();`. A direct cast from `b` to `D` does not compile, because `D` is neither an ancestor nor a descendant of `B`, the static type of `b`.

Jackie then completed the chained version:

```
D d = (D)((A) b);
```

The inner cast `(A) b` compiles as an upward cast from `B` to `A`. The result of that cast expression has static type `A`, so the outer cast to `D` compiles as a downward cast. This explains why the line can pass the compiler.

3. Why the chained cast still fails at run time

Passing compilation is not the end of the analysis. At run time, the dynamic type of `b` is still `C`. The inner cast to `A` succeeds because `C` is a descendant of `A`; the cast creates an alias view but does not change the actual object or its dynamic type.

The outer cast to `D` then asks whether the same dynamic object, still of type `C`, can fulfill the expectation of `D`. It cannot, because `C` is not a descendant of `D`. Therefore the expression compiles but throws `ClassCastException` at the second cast. Jackie used this to warn students that chained casts can delay a compile-time problem into a run-time problem and should not be written casually in real code.

4. The `instanceof` operator

The main new construct was `instanceof`. It has the form

```
x instanceof Y
```

where `x` is an expression denoting an object, such as a reference variable or an array element, and `Y` is a class name. The result is a Boolean value.

For compilation, Jackie connected `instanceof` directly to casting: `x instanceof Y` compiles if and only if the corresponding cast `(Y) x` compiles. In other words, for a single ordinary check, the target class `Y` must be reachable from the static type of `x` by an upward or downward cast.

5. Run-time meaning of `instanceof`

At run time, `instanceof` uses dynamic type. Jackie gave three equivalent ways to read the true case: the dynamic type of `x` can fulfill the expectation of `Y`; the dynamic type of `x` is a descendant of `Y`; or `Y` is an ancestor of the dynamic type of `x`. If none of these holds, the expression evaluates to false.

The key contrast is that `instanceof` does not throw `ClassCastException`. Once it compiles, it evaluates to either `true` or `false`. This makes it useful as a run-time safety check before a cast.

6. Comparing `instanceof` with a cast

Jackie organized the comparison in a table. Assuming both expressions compile, instanceof is true exactly when the corresponding cast would succeed. If the dynamic type cannot fulfill the target type, the check is false; the cast throws an exception.

This gives the practical guard pattern:

```
if (x instanceof Y) {
  Y y = (Y) x;
}
```

The check and cast should be close together. Jackie warned that if the dynamic type of the reference changes between the check and the cast, the old check no longer guarantees safety.

7. Abstract hierarchy: static type versus dynamic type regions

The next diagram used a generic declaration `A obj = new B();`. Because this declaration compiles, `B` is a descendant of `A`. Jackie then separated the classes in the hierarchy into regions.

Relative to the static type `A`, classes that are ancestors or descendants of `A` make `obj instanceof ??` and `(??) obj` compile. Classes that are neither ancestors nor descendants of `A` fail at compilation. Among the compilable cases, the classes that are ancestors of the dynamic type `B` give `instanceof` true and a successful cast; the other compilable classes give `instanceof` false and would cause `ClassCastException` if cast directly.

8. Student hierarchy example: bypassing an unsafe cast

The student example used `Student jim = new NonResidentStudent(...)` and then tested whether `jim instanceof ResidentStudent`. The corresponding cast `(ResidentStudent) jim` compiles because it is a downward cast from the static type `Student`. However, it would fail at run time because the dynamic type `NonResidentStudent` cannot fulfill the expectation of `ResidentStudent`.

With the `instanceof` guard, the condition evaluates to `false`, so the branch containing the cast and resident-specific method call is bypassed. This is the point of the guard: it prevents the unsafe cast from executing.

9. Smartphone `instanceof` drills

The smartphone examples used `SmartPhone myPhone = new Samsung();`. Since the static type is `SmartPhone`, checks against descendant classes such as `Android`, `Samsung`, `GalaxyS21`, `IOS`, and `IPhone13Pro` all compile as possible downcast-style checks.

At run time, however, the dynamic type is `Samsung`. Therefore `myPhone instanceof Android` and `myPhone instanceof Samsung` evaluate to `true`, while checks against classes outside the ancestor chain of `Samsung`, such as `GalaxyS21`, `IOS`, and `IPhone13Pro`, evaluate to `false`. Jackie emphasized precomputing the ancestors of the dynamic type before answering these questions.

10. Safe casts through guarded branches

Jackie then combined `instanceof` and casts in short `if` statements. If the Samsung check is true, casting `myPhone` to `Samsung` is safe. If checks for other descendant classes are false, their branches are skipped, so the risky casts inside them are never executed.

This segment is important because it turns the abstract rules into a programming habit: do not rely on wishful thinking for downcasts; check the dynamic type and cast only when the check succeeds.

11. Static type still controls method-call compilation

The lecture then returned to method availability. If `sp` has static type `SmartPhone`, the calls expected from `SmartPhone`, such as `dial()` and `surfWeb()`, compile. Subclass-only calls such as `FaceTime()` and `quickTake()` do not compile through `sp`.

Changing only the static type changes what can be called. With static type `IOS`, `dial()` and `FaceTime()` compile, but `quickTake()` still does not. With static type `IPhone13Pro`, all three compile. The dynamic object may be the same in all cases, but the compiler uses the static type to decide which method calls are legal.

12. Anonymous casts for subclass-only methods

The anonymous cast examples showed how a cast changes the static type of the expression immediately before the dot. For example, if `sp` has static type `SmartPhone` but points to an `IPhone13Pro`, then `sp.FaceTime()` does not compile, but

```
((IPhone13Pro) sp).FaceTime();
```

can compile because the context object to the left of the dot now has the cast type as its static type. Run-time safety still depends on the actual dynamic type.

13. Call by value and inheritance

The final few minutes recalled call by value. In a method call `obj.m(arg)`, the parameter receives the argument value as if an implicit assignment occurred:

```
param = arg;
```

If the parameter has static type `T1` and the argument has static type `T2`, then the invocation compiles only when `T2` can fulfill `T1`'s expectation; equivalently, `T2` is a descendant of `T1`. Jackie asked students to review this page before the next class, where the idea would be developed further.

Key Terms, Notation, and Structures

- **Static type**: the declared type used by the compiler for method-call legality, cast compilation, `instanceof` compilation, and assignment compatibility.
- **Dynamic type**: the actual run-time class of the object currently referenced; it controls cast success, `instanceof` truth, and dynamic dispatch.
- **Cast type**: the class name inside a cast such as `(Y) x`; the cast expression has static type `Y` if it compiles.
- **Upward cast**: a cast to an ancestor of the expression's static type; it reduces expectation and is normally safe with respect to dynamic type.
- **Downward cast**: a cast to a descendant of the expression's static type; it increases expectation and can throw `ClassCastException`.
- **Chained cast**: a sequence such as `(D)((A) b)` that makes each individual cast legal even when the direct cast would not compile.
- `instanceof`: a Boolean operator of the form `x instanceof Y`; it compiles when `(Y) x` compiles and evaluates according to the dynamic type of `x`.
- **ClassCastException**: the run-time exception thrown when a cast executes but the dynamic type cannot fulfill the expectation of the cast type.
- **Guarded cast**: the pattern of checking `x instanceof Y` before performing `(Y) x`.
- **Expectation**: the range of methods and attributes callable through a given static type.
- **Rule of substitution**: for `param = arg` or ordinary assignment, the right-hand side static type must fulfill the left-hand side static type, usually by being the same type or a descendant.

Key Examples and Demonstrations

1. Chained cast through `A`

Setup: In the symbolic hierarchy, `B extends A`, `C extends B`, and `D extends A`. The variable `b` has static type `B` and dynamic type `C`.

Main point: `(D)((A) b)` compiles because `(A) b` is an upcast and the outer `(D)` is then a downcast from static type `A`. It still throws `ClassCastException` at the outer cast because the dynamic type remains `C`, and `C` is not a descendant of `D`.

What to practise: For every chained cast, analyze each cast twice: once for compilation using static type, and once for run-time safety using dynamic type.

2. Meaning of `x instanceof Y`

Setup: `x` is a reference expression and `Y` is a class name.

Main point: The expression compiles if `(Y) x` compiles. At run time, it is true exactly when the dynamic type of `x` is `Y` or a descendant of `Y`.

What to practise: Draw a hierarchy, mark `x`'s static and dynamic types, and decide separately whether the expression compiles and whether it evaluates to true.

3. Guarding a resident-student cast

Setup: `Student jim = new NonResidentStudent(...);` followed by `if (jim instanceof ResidentStudent) { ... }`.

Main point: The check compiles but evaluates to `false`; therefore the branch containing `(ResidentStudent) jim` is bypassed. Without the guard, that cast would throw `ClassCastException`.

What to practise: Explain why the cast compiles but is not exception-free, and identify exactly why the `if` statement prevents the exception.

4. Smartphone truth values

Setup: `SmartPhone myPhone = new Samsung();`.

Main point: With static type `SmartPhone`, many `instanceof` checks compile because the target classes are descendants of `SmartPhone`. With dynamic type `Samsung`, only checks against `Samsung` or its ancestors evaluate to `true`.

What to practise: List the ancestors of the dynamic type before deciding run-time results; do not let the broad static type mislead you.

5. Static type and method calls

Setup: The same phone object is referenced using static types `SmartPhone`, `IOS`, and the specific iPhone type.

Main point: The more specific the static type, the larger the callable method set. Basic phone types expose basic methods; `IOS` adds `FaceTime()`; the specific iPhone type adds `quickTake()`.

What to practise: For each variable declaration, ignore the right-hand side first and list the methods available from the static type alone.

6. Parameter passing as substitution

Setup: A method has parameter type `T1`; the caller passes an argument of static type `T2`.

Main point: Call by value behaves like the implicit assignment `param = arg`. Therefore the call is type-correct only when `T2` can fulfill `T1`'s expectation.

What to practise: Rewrite method calls as implicit assignments and apply the same substitution rule used for variable assignments.

Common Pitfalls and Misconceptions

1. Thinking a chained cast is safe because it compiles

Pitfall: Treating `(D)((A) b)` as safe once both individual casts pass the compiler.

Why it causes problems: The dynamic type is still the original object's type. If that dynamic type cannot fulfill the final cast type, the outer cast throws `ClassCastException`.

How to avoid it: After every cast, remember that the static type of the cast expression may change, but the dynamic type of the object does not.

2. Using static type to decide `ClassCastException`

Pitfall: Asking whether the original static type is a descendant of the cast type when deciding run-time safety.

Why it causes problems: Static type is for compilation. Run-time cast safety depends on the dynamic type.

How to avoid it: Use the question: can the dynamic type fulfill the expectation of the cast type?

3. Treating `instanceof` like a cast

Pitfall: Expecting `instanceof` to throw an exception when the dynamic type is incompatible.

Why it causes problems: `instanceof` is a Boolean check. If it compiles, it evaluates to `true` or `false`; it does not create a casted alias.

How to avoid it: Use `instanceof` to decide whether to perform a cast, but remember that the cast is still a separate operation.

4. Forgetting that `instanceof` also has a compilation rule

Pitfall: Assuming any `x instanceof Y` expression is legal as long as it can be evaluated at run time.

Why it causes problems: Java still checks whether the analogous cast `(Y) x` is meaningful from the static type.

How to avoid it: First ask whether `(Y) x` would compile, then ask whether the dynamic type makes the condition true.

5. Moving or changing the reference between check and cast

Pitfall: Checking `x instanceof Y`, changing `x`, and then relying on the old check to justify `(Y) x`.

Why it causes problems: The guarantee depends on the same dynamic object still being referenced.

How to avoid it: Keep the guarded cast immediately after the check, or re-check after any reassignment.

6. Letting dynamic type add methods at compile time

Pitfall: Saying `sp.FaceTime()` compiles because `sp` currently points to an `IPhone13Pro` object.

Why it causes problems: Method-call compilation is based on the static type to the left of the dot.

How to avoid it: To access subclass-only methods, either use a sufficiently specific static type or use a correctly guarded cast.

7. Ignoring call-by-value as an assignment

Pitfall: Treating parameter passing as separate from the rule of substitution.

Why it causes problems: A method call with an incompatible argument type fails for the same reason an incompatible assignment fails.

How to avoid it: Translate the call into `param = arg` and check whether the argument type fulfills the parameter type.

What You Should Be Able to Do

After this lecture, you should be able to:

- Draw the inheritance hierarchy from `extends` declarations before analyzing casts or `instanceof` checks.
- Explain why `(D) b` fails to compile when `D` is neither an ancestor nor a descendant of `B`.
- Explain why `(D)((A) b)` compiles but can still throw `ClassCastException`.
- Trace a chained cast by identifying the static type and dynamic type after each cast expression.
- State the syntax and meaning of `x instanceof Y`.
- Determine whether an `instanceof` expression compiles by checking whether the analogous cast compiles.
- Determine whether an `instanceof` expression evaluates to `true` by comparing the dynamic type of `x` with `Y`.
- Use `instanceof` as a guard before a downcast.
- Explain why a false `instanceof` condition safely bypasses an unsafe cast inside an `if` branch.

- Compare `instanceof` with a cast: Boolean result versus alias creation or a cast exception.
- Classify target classes as non-compilable, compilable and true, compilable and false, exception-free, or exception-throwing as appropriate.
- Predict which method calls compile based on a variable's static type.
- Use anonymous casts with correct parentheses when accessing subclass-only methods.
- Model call by value as `param = arg` and apply the rule of substitution to method arguments.
- Separate Written Test 2 coverage from later final-exam material according to Jackie's announcement.

Suggested Practice

- Redraw the `A/B/C/D` hierarchy and explain both why `(D) b` fails to compile and why `(D)((A) b)` throws at run time.
- For the chained-cast example, write one sentence for each cast: whether it compiles, whether it succeeds at run time, and which type justifies the answer.
- Practise `instanceof` checks by marking static type and dynamic type in the hierarchy before answering.
- Re-solve the student example with `Student jim = new NonResidentStudent(...)` and explain why the branch guarded by `jim instanceof ResidentStudent` is bypassed.
- Re-solve the smartphone examples with `SmartPhone myPhone = new Samsung();;` list which checks compile and which evaluate to `true`.
- Write small Java snippets that compare an unsafe cast with the guarded version using `instanceof`.
- For pages 11–14 of Jackie's iPad notes, ignore dynamic types first and decide which method calls compile from static type alone.
- Practise anonymous casts such as `((IPhone13Pro) sp).FaceTime();` and be ready to explain both the compilation check and the run-time check.
- Before the next class, review the call-by-value page and rewrite method calls as implicit `param = arg` assignments.
- For Written Test 2, focus on the casting exercise Jackie identified as in scope; save the new `instanceof` and call-by-value details for final-exam preparation unless Jackie later says otherwise.

Where to Review

- **iPad notes page 1**: Lecture title and the three targets: `instanceof` versus compilable casts, `instanceof` versus `ClassCastException` freedom, and call by value with inheritance.
- **iPad notes page 2**: Final exam, Written Test 2, Lab 4, and Programming Test 3 reminders.
- **iPad notes page 3**: Chained-cast follow-up using the symbolic `A/B/C/D` hierarchy.
- **iPad notes page 4**: Definition of `instanceof`, its compilation condition, and its run-time true condition.
- **iPad notes page 5**: Table comparing `x instanceof Y` with `(Y) x` when both expressions compile.
- **iPad notes page 6**: Abstract hierarchy showing which targets compile, which evaluate true, and which casts succeed or throw.
- **iPad notes pages 7–8**: Student and smartphone dynamic-type checks at run time.
- **iPad notes pages 9–10**: `instanceof` truth-value drills and safe-cast branches in the smartphone hierarchy.
- **iPad notes pages 11–14**: Static type, casts, and polymorphism examples showing which method calls compile.
- **iPad notes page 15**: Student example using `instanceof` and anonymous casts around resident-specific behaviour.
- **iPad notes page 16**: Call-by-value diagram and the implicit assignment `param = arg`.
- **Lecture recording**: Review the opening assessment clarification, the chained-cast run-time trace, the `instanceof` definition, and the call-by-value bridge at the end.
- **Practice materials**: Use the Written Test 2 guide and example questions for in-scope casting practice, then return to slides 68–71 for `instanceof` practice after the test.

Closing Reminder

This digest is only a roadmap for Lecture 20. The important work is to revisit Jackie's iPad notes, recording, examples, and practice materials, then actively draw hierarchies, mark static and dynamic types, and explain each compile-time and run-time decision. Reading the digest alone is not enough preparation for Written Test 2, Programming Test 3, or the final exam.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 21, July 21

Casting Arguments, Polymorphic Arrays, and Polymorphic Return Types

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 6

Key Examples and Demonstrations 7

Common Pitfalls and Misconceptions 9

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 12

Main Ideas

- The lecture connected inheritance to method calls by treating call by value as an implicit assignment from argument to parameter.
- Jackie contrasted `getClass()` with `instanceof`: `getClass()` checks exact dynamic type equality, while `instanceof` allows descendant dynamic types.
- The rule of substitution was applied to method arguments: a call compiles only when the argument's static type can fulfill the parameter's static type.
- The `StudentManagementSystem` example showed how parameter choices affect flexibility: `addRS(ResidentStudent rs)` is restrictive, while `addStudent(Student s)` is broadly usable.
- Explicit casts can make otherwise invalid method calls compile, but the cast still has to pass the run-time dynamic-type check.
- A cast used as a method argument should be analyzed in two stages: first the cast expression, then the method-call substitution.
- A `Student[]` array can be used as a polymorphic collection whose elements have static type `Student` but different dynamic types.
- `instanceof` becomes especially useful when iterating through a polymorphic array and checking the dynamic type at each index.
- A method can have a static return type such as `Student` while returning objects whose dynamic types are `ResidentStudent` or `NonResidentStudent`.
- Dynamic dispatch explains why the same call, such as `s.getTuition()`, can execute different method versions after `s` is reassigned to objects from different array positions.
- Jackie reminded students that Lab 4 was due on Wednesday, July 22, and that Programming Test 3 was scheduled for July 27 with guide and example questions released.

Roadmap

1. Course status and assessment reminders

Jackie opened by saying that the hardest parts of inheritance had already been covered: the rule of substitution, `instanceof`, and type casting. The remaining inheritance material was mostly about applying those ideas. The plan was to finish inheritance soon and then move to recursion, which Jackie described as a bridge toward later data structures material.

The final exam date and time remained Monday, August 10, from 9:00 AM to 12:00 noon, with the room still to be announced. Students were reminded that Lecture Digests were posted. Programming Test 3 was coming on July 27, and Jackie told students to focus strongly on Lab 4, whose solution and submission deadline were both associated with

Wednesday, July 22. Jackie also mentioned that Programming Test 2 automated grading was done, with some manual grading still pending.

2. Comparing `getClass()` and `instanceof`

The lecture began technically by revisiting a student question about `getClass()` versus `instanceof`. In the equality lecture, `x.getClass() == y.getClass()` was used to check whether two objects have exactly the same dynamic type. There is no inheritance flexibility in that comparison.

By contrast, `x instanceof Y` evaluates to true when the dynamic type of `x` can fulfill the expectation of `Y`; equivalently, the dynamic type of `x` is `Y` itself or a descendant of `Y`. Jackie illustrated the important mixed case: if `Y` is an ancestor of `x`'s dynamic type, then `x.getClass() == Y` would be false, but `x instanceof Y` would be true.

A student asked whether downcasting could make a `getClass()` comparison become true. Jackie clarified that a cast creates an alias with a different static type; it does not change the underlying object's dynamic type. Therefore, casting does not make `getClass()` report a different dynamic type.

3. Call by value as hidden substitution

Jackie then returned to call by value. A method call such as `obj.m(arg)` can be understood as an implicit assignment from the argument to the parameter before the method body runs:

```
param = arg;
```

After inheritance is introduced, that hidden assignment must obey the same rule of substitution as ordinary assignments. If the parameter has static type `T1` and the argument has static type `T2`, then the call compiles only when `T2` can fulfill `T1`'s expectation; equivalently, `T2` is `T1` itself or a descendant of `T1`.

This principle is the backbone of the rest of the lecture. Whenever a method call is analyzed for compilation, the dynamic type should be ignored and the static types of the parameter and argument should be compared.

4. Static types in `StudentManagementSystem`

The main example used a `StudentManagementSystem` class with an array

```
Student[] ss;
```

and three methods with different parameter types:

```
void addRS(ResidentStudent rs)
void addNRS(NonResidentStudent nrs)
void addStudent(Student s)
```

The static type of every expression `ss[i]` is `Student`, because the array is declared as `Student[]`. This is true even when a particular element later points to a `ResidentStudent` or `NonResidentStudent` object dynamically.

Inside `addRS`, the assignment `ss[c] = rs` compiles because the right-hand side has static type `ResidentStudent`, which is a descendant of the left-hand side static type `Student`. Jackie warned students not to call this a downcast: there is no explicit cast in the assignment. It is just an application of the rule of substitution.

5. Which `addRS` calls compile?

Jackie then asked under what condition a method call such as

```
sms.addRS(o);
```

would compile. Since `addRS` has a parameter of static type `ResidentStudent`, call by value gives the hidden assignment `rs = o`. Therefore, the static type of `o` must be `ResidentStudent` or a descendant of `ResidentStudent`.

In the worked examples, variables `s1`, `s2`, and `s3` all had static type `Student`, even though some of them had more specific dynamic types. Calls such as `sms.addRS(s2)` still did not compile because the compiler only sees the static type `Student`. The call with `rs`, whose static type is `ResidentStudent`, compiled. The call with `nrs`, whose static type is `NonResidentStudent`, did not compile because `NonResidentStudent` is not a descendant of `ResidentStudent`.

By contrast, calls to `addStudent(Student s)` are much more flexible because `Student` is at the top of this small hierarchy. Any argument whose static type is `Student`, `ResidentStudent`, or `NonResidentStudent` can fulfill the parameter's expectation.

6. Casting method arguments

The lecture next considered how a cast can be used to make a method call compile. For example,

```
sms.addRS((ResidentStudent) s);
```

can compile even when `s` has static type `Student`. The analysis has two parts. First, the cast `(ResidentStudent) s` compiles as a downward cast from `Student` to `ResidentStudent`. Second, the resulting cast expression has static type `ResidentStudent`, so it can be passed to `addRS`.

Run time is a separate question. If `s`'s dynamic type is just `Student`, the cast to `ResidentStudent` throws `ClassCastException`. If `s`'s dynamic type is `NonResidentStudent`, the cast also throws, because `NonResidentStudent` cannot fulfill `ResidentStudent`. If `s`'s dynamic type is `ResidentStudent`, the cast succeeds and the method call can proceed. This example synthesized the lecture's earlier distinction between static type for compilation and dynamic type for run-time safety.

7. Polymorphic arrays

The next major design idea was a polymorphic collection. Jackie contrasted the pre-inheritance design, which needed separate arrays for different kinds of students, with the inheritance-based design using a single `Student[]` array. A `Student[]` array has a fixed static element type, but its elements can dynamically point to `ResidentStudent`, `NonResidentStudent`, or other descendant objects.

The example created a `ResidentStudent` named Rachel and a `NonResidentStudent` named Nancy, then added both through `addStudent`. These calls compiled because the static types `ResidentStudent` and `NonResidentStudent` both fulfill the parameter type `Student`. After the calls, `ss[0]` and `ss[1]` still have static type `Student`, but their dynamic types differ.

Jackie also traced a `registerAll` loop to show that the same array can support common operations on all students, regardless of the dynamic type at each index.

8. Using `instanceof` on a polymorphic array

Because a polymorphic array hides different dynamic types behind the same static type, `instanceof` is used to inspect what each element is dynamically. In the example, `ss[0]` pointed to a `ResidentStudent`, so:

`ss[0] instanceof NonResidentStudent` evaluates to `false`;

`ss[0] instanceof ResidentStudent` evaluates to `true`;

`ss[0] instanceof Student` evaluates to `true`.

For `ss[1]`, whose dynamic type was `NonResidentStudent`, the corresponding answers were reversed for the two subclasses but remained true for `Student`. The important study habit is to locate which index is being accessed, identify that element's dynamic type, and then evaluate each `instanceof` condition accordingly.

9. Polymorphic return types and dynamic dispatch

The lecture ended with a method such as

```
Student getStudent(int i)
```

whose static return type is `Student`. Since the method returns an element from a polymorphic array, the returned object's dynamic type depends on which index is accessed. For index 0, the returned object in the example was dynamically a `ResidentStudent`; for index 1, it was dynamically a `NonResidentStudent`.

When the returned value is assigned to a variable `s` of static type `Student`, a call such as `s.getTuition()` compiles because `getTuition` is part of `Student`'s expectation. At run time, dynamic dispatch chooses the version based on the dynamic type currently stored in `s`: the `ResidentStudent` version for the object at index 0 and the `NonResidentStudent` version for the object at index 1.

Jackie finished by pointing out a common compilation trap. Even if `sms.getStudent(1)` dynamically returns a `NonResidentStudent`, the expression's static type is still the declared return type `Student`. Therefore,

```
NonResidentStudent nrs2 = sms.getStudent(1);
```

does not compile without an explicit cast, because `Student` is not a descendant of `NonResidentStudent`.

Key Terms, Notation, and Structures

`getClass()`
: A method used here for exact dynamic type comparison; `x.getClass() == y.getClass()` is true only when the two dynamic types are identical.

`instanceof`
: A Boolean operator that checks whether an object's dynamic type can fulfill the expectation of the class named on the right-hand side.

Static type
: The declared type of a variable, parameter, array element expression, cast expression, or method return expression; this is what the compiler uses.

Dynamic type
: The actual run-time class of the object being referenced; this is what matters for `instanceof`, dynamic dispatch, and whether a cast throws.

Call by value
: The method-call mechanism modeled in this lecture as the hidden assignment `param = arg`.

Rule of substitution
: For an assignment or hidden parameter assignment to compile, the right-hand side or argument static type must be the same as, or a descendant of, the left-hand side or parameter static type.

`StudentManagementSystem`
: The main example class containing a `Student[]` array, a counter, and methods for adding or retrieving students.

Polymorphic array
: A collection such as `Student[] ss` whose element expressions have one static type but may point to objects of different descendant dynamic types.

Polymorphic return value
: A return value whose declared static type is an ancestor type, even though the returned object can dynamically be one of several descendants.

Dynamic dispatch
: The run-time selection of the method implementation according to the dynamic type of the object, as in the `getTuition()` examples.

Cast expression
: An expression such as `(ResidentStudent) s`; if successful, it creates an alias with the cast type as its static type.

`ClassCastException`
: A run-time exception thrown when a compilable cast is applied to an object whose dynamic type cannot fulfill the cast type's expectation.

Important code patterns from the lecture include:

- `x.getClass() == y.getClass()` for exact dynamic type equality.
- `x instanceof Y` for inheritance-aware dynamic type checking.
- `param = arg` as the call-by-value model.
- `ss[c] = rs;` as an ordinary substitution into a `Student[]` array, not a cast.
- `sms.addRS((ResidentStudent) s);` as a method call whose argument is a cast expression.
- `Student s = sms.getStudent(i);` as a polymorphic return assigned to an ancestor-typed variable.

Key Examples and Demonstrations

1. Exact versus inheritance-aware dynamic type checks

Jackie compared `x.getClass() == y.getClass()` with `x instanceof Y`. If `Y` is exactly `x`'s dynamic type, both ideas can report success. If `Y` is an ancestor of `x`'s dynamic type, `getClass()` equality fails because the types are not identical, while `instanceof` succeeds because the dynamic type can fulfill `Y`'s expectation.

The key demonstration point was that casting does not change this: a cast changes the static type of a reference expression, not the dynamic type reported by `getClass()`.

2. Compiling `addRS` calls

For `void addRS(ResidentStudent rs)`, the hidden call-by-value assignment is `rs = argument`. Therefore, only arguments whose static types are `ResidentStudent` or descendants can be passed without a cast.

This explains why variables declared as `Student` do not work as arguments to `addRS`, even when their dynamic types are more specific. It also explains why `NonResidentStudent nrs` does not work: it is not a descendant of `ResidentStudent`.

3. Casting an argument to satisfy the compiler

The call

```
sms.addRS((ResidentStudent) s);
```

illustrates a two-stage analysis. First, the cast from `Student` to `ResidentStudent` is a downward cast, so the cast expression compiles. Second, the cast expression has static type `ResidentStudent`, so it can be passed to `addRS`.

The run-time result depends on the dynamic type of `s`. If `s` points to a plain `Student` or a `NonResidentStudent`, the cast throws `ClassCastException`. If `s` points to a `ResidentStudent`, the cast succeeds.

4. Building a polymorphic array

The lecture used Rachel as a `ResidentStudent` and Nancy as a `NonResidentStudent`. Both could be added to the same `Student[]` array through `addStudent(Student s)` because their static types are descendants of `Student`.

After the calls, `ss[0]` and `ss[1]` are still expressions of static type `Student`, but they point to objects with different dynamic types. This is the central example of a polymorphic array.

5. Tracing `instanceof` through array indices

The `for` loop over the array made students track the current index. At index 0, the dynamic type was `ResidentStudent`; at index 1, the dynamic type was `NonResidentStudent`. Each `instanceof` result followed from that dynamic type and the hierarchy.

This demonstration is important because the code text `ss[i]` always has the same static type, but the actual object can change from iteration to iteration.

6. Returning from a polymorphic array

The accessor `getStudent(int i)` had static return type `Student`, but returned `ss[i]`. If `i` is 0, the returned object was dynamically a `ResidentStudent`; if `i` is 1, it was dynamically a `NonResidentStudent`.

When the result was stored in `Student s`, `s.getTuition()` could call different overridden versions depending on the current dynamic type. This was the lecture's main dynamic-dispatch demonstration.

7. The failed subclass assignment from a polymorphic return

The final warning example was:

```
NonResidentStudent nrs2 = sms.getStudent(1);
```

Even though index 1 happened to store a `NonResidentStudent` dynamically, the compiler only sees the method's static return type `Student`. Since `Student` is not a descendant of `NonResidentStudent`, the assignment does not compile without a cast.

Common Pitfalls and Misconceptions

- **Using dynamic type to answer a compilation question.** If Jackie asks whether a method call compiles, ignore dynamic type and compare the relevant static types.
- **Reversing the parameter-argument direction.** For `obj.m(arg)`, think `param = arg`; the argument's static type must fulfill the parameter's static type, not the other way around.
- **Calling an ordinary assignment a downcast.** In `ss[c] = rs`, there is no cast syntax; the assignment compiles by the rule of substitution.
- **Thinking a cast changes dynamic type.** A cast creates an alias with a different static type; the object itself and its `getClass()` result do not change.
- **Assuming `sms.addRS(s2)` compiles because `s2` dynamically points to a resident student.** The compiler sees `s2`'s static type `Student`, so the call still fails.
- **Forgetting the two-stage analysis of casted arguments.** First check whether the cast expression compiles; then check whether the resulting static type satisfies the method parameter.
- **Assuming a casted method call is run-time safe.** A cast can make the compiler accept the call but still throw `ClassCastException` before the method body executes.
- **Confusing the static type of `rs` with the static type of `ss[0]`.** The variable `rs` keeps static type `ResidentStudent`; the expression `ss[0]` has static type `Student` because of the array declaration.
- **Expecting the compiler to know which array index is used.** Even if `sms.getStudent(1)` dynamically returns a `NonResidentStudent`, its static return type is still `Student`.
- **Treating polymorphic arrays as merely a coding shortcut.** The design point is deeper: one ancestor-typed collection can hold multiple descendant dynamic types while still supporting common operations.

What You Should Be Able to Do

After this lecture, you should be able to:

- Explain the difference between exact `getClass()` comparison and inheritance-aware `instanceof` checking.
- Explain why casting does not change an object's dynamic type.
- Model a method call as call by value using the hidden assignment `param = arg`.
- Apply the rule of substitution to decide whether a method call compiles.

- Identify the static types of method parameters, arguments, array element expressions, and method return expressions.
- Determine why `sms.addRS(s1)`, `sms.addRS(s2)`, `sms.addRS(s3)`, and `sms.addRS(nrs)` fail to compile in the lecture setup.
- Explain why `sms.addRS(rs)` compiles.
- Explain why the method `addStudent(Student s)` accepts more argument types than the method `addRS(ResidentStudent rs)`.
- Analyze a method call whose argument is a cast expression, separating compilation from run-time `ClassCastException` analysis.
- Explain what a polymorphic array is and why `Student[] ss` can store different descendant dynamic types.
- Trace a loop over a polymorphic array and evaluate `instanceof` checks at each index.
- Distinguish the static type of `ss[i]` from the dynamic type of the object stored at that index.
- Explain how a method with static return type `Student` can return different dynamic types.
- Trace dynamic dispatch for `getTuition()` after retrieving different objects from a polymorphic array.
- Justify why assigning `sms.getStudent(1)` directly to a `NonResidentStudent` variable does not compile without a cast.

Suggested Practice

- Recreate Jackie's comparison of `getClass()` and `instanceof` using a small hierarchy where one class is an ancestor of another.
- For each `addRS` example from the notes, write the hidden call-by-value assignment and identify both static types.
- Explain in one sentence why each of `s1`, `s2`, and `s3` fails as an `addRS` argument despite possible dynamic-type differences.
- Rework the casting-argument examples on slides 79–82: first check compilation, then check possible `ClassCastException` at run time.
- Compare the no-inheritance design on slide 11 with the polymorphic collection design on slide 83 and explain why the latter needs only one array.
- Redraw the `Student`, `ResidentStudent`, `NonResidentStudent` hierarchy and mark the static and dynamic types for `ss[0]` and `ss[1]`.

- Trace the `instanceof` loop twice: once for index 0 and once for index 1.
- Trace `s = sms.getStudent(0); s.getTuition();` and `s = sms.getStudent(1); s.getTuition();`, naming the method version selected in each case.
- Modify the polymorphic return example by adding an explicit cast before assigning to `NonResidentStudent`, then analyze when it would be safe.
- For Programming Test 3, focus on Lab 4, review the released guide and example questions, submit Lab 4 on time, and study the solution after it is released.

Where to Review

- **iPad notes page 1**: Lecture title and the three main topics: call by value with inheritance, casting method arguments, and polymorphic arrays and return types.
- **iPad notes page 2**: Lab 4 due date, Programming Test 3 date, guide/example-question reminder, and final exam reminder.
- **iPad notes page 3**: `getClass()` versus `instanceof`, including the exact-type versus ancestor-type distinction.
- **iPad notes page 4**: Polymorphic parameter setup in `StudentManagementSystem`; static type of `ss[i]` and the rule for `sms.addRS(o)`.
- **iPad notes page 5**: Examples deciding which `addRS` and `addStudent` calls compile.
- **iPad notes page 6**: Casting method arguments and the three dynamic-type cases for `ClassCastException`.
- **iPad notes page 7**: Polymorphic collection example; compare slide 83 with slide 11 and trace `instanceof` results for `ss[0]` and `ss[1]`.
- **iPad notes page 8**: Polymorphic return type example, dynamic dispatch through `getTuition()`, and the failed subclass assignment from a `Student` return type.
- **Lecture recording**: Review Jackie's explanation of why dynamic type is irrelevant for compilation, the casted-argument examples, and the polymorphic array trace.
- **Slides**: Revisit slide 11 versus slide 83 for the design comparison, slide 78 for the setup of the method-call examples, and slides 79–82 for casting arguments.
- **Lab and test materials**: Use Lab 4, the Programming Test 3 guide, and the released example questions as the main near-term programming-test preparation materials.

Closing Reminder

This digest is only a roadmap for Lecture 21. The important work is to revisit Jackie's iPad notes, recording, examples, Lab 4 materials, and practice questions, then actively trace static types, dynamic types, hidden parameter assignments, casts, array indices, and returned objects. Reading the digest alone is not enough preparation for Programming Test 3 or the final exam.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 22, July 23

Dynamic Binding & Overridden `equals`, Recursion: Principles & Tracing

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 6

Key Examples and Demonstrations 8

Common Pitfalls and Misconceptions 10

What You Should Be Able to Do 11

Suggested Practice 12

Where to Review 13

Closing Reminder 14

Main Ideas

- The lecture closed the inheritance unit by synthesizing three habits: know the Java syntax, judge compilation using static types, and predict run-time behaviour using dynamic types.
- Jackie reviewed polymorphic arrays and polymorphic return values as major applications of inheritance: one static supertype can represent many dynamic subtype objects.
- Dynamic binding was revisited through `equals`: when an overridden method is called, Java starts from the receiver's dynamic type and chooses the nearest available implementation along the ancestor path.
- The type-checking table on slide 90 was presented as a compact study guide for assignments, method calls, accessors, casts, and method calls with cast arguments.
- The lecture then introduced recursion as the final major course topic, with the exam focus on direct recursive Java methods for strings and arrays.
- A recursive solution was framed as base cases for small problems and recursive cases that divide a big problem into strictly smaller subproblems, solve them recursively, and combine the results.
- Factorial was used to show the difference between a non-recursive formula and a recursive definition with a base case and a strictly smaller recursive call.
- Jackie demonstrated two ways to trace recursion: a run-time stack for precise method-call execution, and a recursion tree for building design intuition.
- The common recursion errors were missing a base case and making a recursive call on a problem that is not strictly smaller; both lead to non-termination and stack overflow.
- Fibonacci numbers were introduced as a second recursive example with two base cases and two recursive calls, and students were asked to trace `fib(4)`.
- For immediate preparation, Jackie reminded students to use the Lab 4 solution walkthrough, redo Lab 4 under time pressure, and prepare for Programming Test 3 on July 27.

Roadmap

1. Logistics and programming-test preparation

Jackie began with assessment and resource reminders. The Lab 4 solution walkthrough had been released under the inheritance section, including a video with timestamps and iPad notes. Jackie recommended prioritizing the inheritance-and-array pattern from Lab 4, then redoing the lab under time pressure. For the practice test, Jackie emphasized that the first attempt is the most valuable because students are not yet programming from memory.

Programming Test 3 was scheduled for Monday, July 27 during the scheduled lab session. A practice-test solution recording was planned for Friday, and Programming Test 2 results were expected later Friday or by Sunday afternoon at the latest. Jackie also said that exam discussion, early tips, and exam-related questions would be handled the following Tuesday, with the last class on Thursday.

2. Inheritance wrap-up: three things to synthesize

Before moving to recursion, Jackie summarized how students should organize the inheritance unit. The first layer is syntax: `super`, `extends`, `instanceof`, and type-cast syntax. The second layer is compilation: use static types to decide whether Java accepts an assignment, method call, return assignment, or cast. The third layer is run-time behaviour: use dynamic types to predict dynamic binding and possible `ClassCastException`s.

This synthesis is important because exam-style questions often mix these layers. A correct answer must say which phase is being analyzed and which type information is relevant at that phase.

3. Polymorphic arrays and return values revisited

Jackie briefly revisited the polymorphic collection example from the previous lecture. A method such as `addStudent(Student s)` is tolerant because `Student` is at the top of the hierarchy; arguments whose static types are descendant classes can be accepted. This makes it possible for a single `Student[]` array to store objects that are dynamically `ResidentStudent`, `NonResidentStudent`, or another descendant.

The same idea applies to polymorphic return values. If an accessor has static return type `Student`, then every call to it is seen by the compiler as returning a `Student`. At run time, however, the object returned from the underlying polymorphic array may have a more specific dynamic type. Students should connect this directly to dynamic dispatch: after such a return, a call such as `s.getTuition()` may execute a subclass version depending on the returned object's dynamic type.

4. Dynamic binding for overridden `equals`

The main inheritance example for this lecture used a chain `Object -> A -> B -> C` and a call such as `c1.equals(c2)`. Jackie asked which `equals` version is called when `c1`'s dynamic type is `C`. The answer depends on where `equals` has been overridden.

If none of `A`, `B`, or `C` overrides `equals`, the default version from `Object` is called. If `C` overrides `equals`, the version in `C` is called directly. If `B` overrides `equals` but `C` does not, Java starts at dynamic type `C`, searches upward, and chooses the overridden version in the closest ancestor, namely `B`. Jackie stressed that students should draw the hierarchy, mark where method versions exist, and choose the closest applicable one.

5. Type-checking rules as an active study table

Jackie then pointed students to slide 90, the type-checking rule summary. The table covers several common code forms: `x = y`, `x.m(y)`, `z = x.m(y)`, `(C) y`, `x = (C) y`, and `x.m((C)`

`y)`. It is dry but complete, so Jackie suggested using it actively: cover the condition column, predict the condition needed for compilation, then reveal and check.

The important pattern is that each row asks a precise compile-time question. For example, an assignment checks whether the right-hand side static type is a descendant of the left-hand side static type. A method call must check that the method is defined for the receiver's static type and that the argument's static type is compatible with the method parameter's static type. Rows involving casts add extra checks about whether the cast type is related to the expression's static type.

6. From compile-time casts to run-time `ClassCastException`

After the table, Jackie assigned an inheritance exercise: once a cast expression is known to compile, under what condition will a `ClassCastException` occur at run time? The reminder was that compile-time permission for a cast is not the same as run-time safety.

The run-time question is always about the dynamic type of the expression being cast. A cast to type `C` fails at run time when the object's dynamic type cannot fulfill the expectation of `C`. Students were directed back to the earlier casting examples as concrete practice.

7. `toString` as another overridden method

Jackie closed the inheritance discussion by mentioning `toString`. Like `equals`, `toString` has a default version inherited from `Object`, and application classes may override it. Although Jackie did not work through it in detail during class, students were asked to study that slide and connect it to the same overridden-method and dynamic-binding reasoning used for `equals`.

8. Recursion resources and exam scope

The lecture then moved to recursion. Jackie presented recursion as a general problem-solving paradigm, not just a Java feature. For this course's exam, the focus is direct recursive Java methods on strings and arrays. Jackie said that CodingBat Recursion 1 is sufficient for exam preparation; CodingBat Recursion 2 is optional enrichment. Jackie also pointed students to a past 2030 recursion lab with questions and solutions, posted Java source code, a tracing exercise for `fib(4)`, and a tutorial section with recordings, notes, and code. Haskell and Tower of Hanoi were mentioned as optional ways to go beyond the exam focus.

9. Base cases and recursive cases

The first recursion principle was the division between base cases and recursive cases. A base case handles a small problem directly, without another recursive call. A recursive case handles a larger problem by dividing it into strictly smaller subproblems.

Jackie described the recursive-case pattern as divide, conquer, and combine: divide a big problem into smaller subproblems, make recursive calls on those subproblems while assuming their returned results are correct, and combine those returned results into a solution for the original problem. The word "strictly" matters: if the subproblem is not strictly smaller, the recursion may never reach a base case.

10. Direct recursion and the role of the argument

Jackie defined direct recursion as a method calling itself. If a method `m` is called on an input represented by `i`, then a recursive call such as `m(j)` should use an argument `j` that represents a strictly smaller problem. In the simplest integer examples, this means replacing `n` by something like `n - 1`. Later, for strings and arrays, the same idea becomes reducing the size or range of the data being processed.

11. Factorial as the first recursive example

Jackie used factorial as the "hello world" example for recursion. The product formula

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

is correct, but it is not recursive because it does not define factorial in terms of a smaller factorial problem.

The recursive definition is:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

The base case is `n == 0`. The recursive case reuses factorial on the strictly smaller input `n - 1`. Jackie connected this definition directly to Java code with an `if` branch for the base case and an `else` branch computing `n * factorial(n - 1)`.

12. Tracing `factorial(3)` with tree and stack

Jackie showed two ways to trace `factorial(3)`. The recursion tree begins from `factorial(3)`, expands to `3 * factorial(2)`, then to `2 * factorial(1)`, then to `1 * factorial(0)`. Once the base case returns 1, the values propagate back upward to return 1, then 2, then 6.

The stack trace is more mechanical. Each recursive call is pushed onto the run-time stack and becomes active, while the previous call is suspended. After `factorial(0)` returns, it is popped; execution resumes at the new top of the stack. Jackie emphasized that when using a stack, students must track the argument value for each activation record because all the method calls have the same method name.

13. Common recursion errors

Two incorrect factorial implementations illustrated the core dangers. The first version had no base case, so `factorial(2)` leads to `factorial(1)`, `factorial(0)`, `factorial(-1)`, and so on. The second version had a base case but recursively called `factorial(n)` instead of `factorial(n - 1)`, so the problem size never decreased.

Jackie corrected the phrase "infinite loop" to "infinite recursion" because no loop construct is involved. In Java, such non-terminating recursion eventually exhausts the run-time stack and causes a stack overflow error.

14. Fibonacci as a second recursive example

The lecture ended by introducing Fibonacci numbers with indices starting at 1:

$$F = 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

The recursive definition has two base cases and one recursive case:

$$F_n = \begin{cases} 1 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ F_{n-1} + F_{n-2} & \text{if } n \geq 3. \end{cases}$$

This example is more complex than factorial because the recursive case makes two recursive calls. Jackie assigned students to trace `fib(4)` using both a stack and a recursion tree before the next class.

Key Terms, Notation, and Structures

Static type
: The declared compile-time type of a variable, expression, parameter, array element expression, or return expression.

Dynamic type
: The actual run-time class of the object being referenced; this governs dynamic binding and run-time cast safety.

Polymorphic array
: An array such as `Student[] ss` whose element expressions have static type `Student` but may point to objects of different descendant dynamic types.

Polymorphic return value
: A returned value whose static return type is an ancestor class even though the actual object returned may have a descendant dynamic type.

Dynamic binding
: The run-time selection of an overridden method implementation according to the dynamic type of the receiver.

Receiver or context object
: The object before the dot in a method call, such as `c1` in `c1.equals(c2)`; its dynamic type determines dynamic binding.

Closest ancestor rule
: If the receiver's dynamic type does not define the method version, Java searches upward and chooses the closest ancestor that provides the method implementation.

`equals(Object obj)`
: The method used in the inheritance wrap-up to demonstrate overridden methods and dynamic binding.

`toString()`
: Another `Object` method that may be overridden; Jackie asked students to study it using the same dispatch reasoning as `equals`.

Type-checking rules
: Compile-time rules for expressions such as `x = y`, `x.m(y)`, `z = x.m(y)`, casts, and calls with cast arguments.

`ClassCastException`
: A run-time exception caused by a cast whose static form compiles but whose target type is not fulfilled by the object's dynamic type.

Recursion
: A problem-solving method where a method solves a problem by calling itself on smaller instances of the same problem.

Direct recursion
: The form emphasized for this course: a method directly calls itself inside its own body.

Base case
: A small-input case solved directly without another recursive call; it is the termination condition.

Recursive case
: A larger-input case that makes one or more recursive calls on strictly smaller subproblems.

Divide, conquer, combine
: The recursive-case structure: split into smaller subproblems, assume recursive calls return correct answers, and combine those answers.

Run-time stack
: The structure used to track active method calls; the top of the stack is the current execution point.

Activation record
: One entry on the run-time stack for an active method call, including its argument values and local state.

Push and pop
: A method call is pushed when activated and popped when it finishes; suspended calls resume when they return to the top of the stack.

Recursion tree
: A diagram of how a recursive call depends on smaller recursive calls; Jackie described it as better for building design intuition than the stack.

Stack overflow
: The failure caused by too many unfinished recursive calls; in Java this corresponds to a `StackOverflowError`.

Important formulas and code shapes from the lecture include:

- `c1.equals(c2)`: dynamic binding starts from `c1`'s dynamic type.
- `x.m(y)`: compile by checking whether `m` is available from `x`'s static type and whether `y`'s static type fits `m`'s parameter type.
- `(C) y`: a cast that compiles when `C` is an ancestor or descendant of `y`'s static type.
- `n * factorial(n - 1)`: the recursive factorial case, using a strictly smaller input.
- `fib(n - 1) + fib(n - 2)`: the recursive Fibonacci case, using two smaller inputs.

Key Examples and Demonstrations

1. Dynamic binding for `equals` in an `A-B-C` chain

Setup: The notes used the hierarchy `Object -> A -> B -> C` and the call `c1.equals(c2)`, where `c1` is dynamically a `C` object. Different slides changed where `equals` was overridden.

Main point: Java chooses the `equals` version by starting at the receiver's dynamic type and searching upward for the nearest implementation. No override in `A`, `B`, or `C` means the `Object` version is called; an override in `C` means `C`'s version is called; an override in `B` with none in `C` means `B`'s version is called.

What to practise: Redraw the hierarchy, mark where each `equals` version exists, circle the receiver's dynamic type, and explain why the closest available version is selected.

2. Reading the type-checking table on slide 90

Setup: Slide 90 summarizes compile-time rules for assignments, method calls, method-return assignments, casts, and method calls with cast arguments.

Main point: Each row combines small checks. For example, `x.m(y)` checks both method availability from `x`'s static type and parameter compatibility for `y`'s static type. A cast expression checks whether the cast type is related to the expression's static type; an assignment or method call that uses the cast then checks the static type produced by the cast.

What to practise: Cover the right column of the table and try to reconstruct each condition. Then create one small `Student` or `SmartPhone` example for each row and justify whether it compiles.

3. `toString` as an independent study example

Setup: Jackie pointed to the `toString` slide after finishing the `equals` dynamic-binding examples.

Main point: `toString` follows the same overriding and dynamic-binding pattern as `equals`: a default version exists in `Object`, but application classes can override it, and Java chooses the appropriate version at run time.

What to practise: Study the `toString` slide and explain it using the same vocabulary used for `equals`: receiver, dynamic type, overridden version, and closest ancestor.

4. Designing a recursive factorial method

Setup: The lecture contrasted the product expression for factorial with the recursive definition

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

Main point: The product expression is correct but not recursive because it does not reuse the factorial problem on a smaller scale. The recursive definition has a base case and a recursive case whose recursive call uses the smaller input `n - 1`.

What to practise: Write the Java version with an `if` base case and an `else` recursive case. Then explain why `factorial(n - 1)` is valid but `factorial(n)` is not.

5. Tracing `factorial(3)` by tree

Setup: Starting from `factorial(3)`, the recursive definition expands to `3 * factorial(2)`, then `2 * factorial(1)`, then `1 * factorial(0)`.

Main point: The recursion tree shows the dependency structure. The computation expands downward until the base case `factorial(0)` returns 1, then values return upward to produce 1, then 2, then 6.

What to practise: Draw the tree without looking at the notes, label which node is the base case, and mark how return values propagate upward.

6. Tracing `factorial(3)` by stack

Setup: The same `factorial(3)` call was traced using the run-time stack.

Main point: Each call `factorial(3)`, `factorial(2)`, `factorial(1)`, and `factorial(0)` is pushed as a separate activation record. After the base case returns, calls are popped in last-in-first-out order. The active call is always the one at the top of the stack.

What to practise: Repeat the stack trace and write the argument value beside each activation record. Do not treat the calls as identical just because they share the same method name.

7. Two incorrect factorial methods

Setup: Jackie showed one factorial method with no base case and another where the recursive call used `factorial(n)` instead of `factorial(n - 1)`.

Main point: The first version never terminates because there is no stopping condition. The second version has a base case but does not move toward it. Both are examples of infinite recursion and eventually exhaust the stack.

What to practise: For each incorrect version, trace the first few recursive calls and identify exactly where the design rule is violated.

8. Fibonacci numbers and `fib(4)`

Setup: The Fibonacci sequence was defined with $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 3$.

Main point: Fibonacci has two base cases and a recursive case with two recursive calls. This makes the trace branch, so it is a natural next step after factorial.

What to practise: Complete Jackie's assigned exercise: trace `fib(4)` using both a recursion tree and a run-time stack before the next class.

Common Pitfalls and Misconceptions

Treating inheritance as isolated facts

Pitfall: Memorizing keywords such as `extends`, `super`, `instanceof`, and casts without connecting them to static-type and dynamic-type reasoning.

Why it causes problems: Exam-style inheritance questions often combine syntax, compilation, dynamic binding, and run-time exceptions in the same code fragment.

How to avoid it: For every inheritance example, label which part is syntax, which question is compile time, and which question is run time.

Choosing an overridden method from the wrong class

Pitfall: Looking at the static type of the receiver, or arbitrarily skipping to a farther ancestor, when deciding which overridden method is called.

Why it causes problems: Dynamic binding uses the receiver's dynamic type and the closest available implementation along the ancestor path.

How to avoid it: Circle the receiver, identify its dynamic type, then search upward until the first matching method version is found.

Thinking a compilable cast is automatically safe

Pitfall: Stopping after proving that a cast compiles.

Why it causes problems: A cast may compile because the cast type is related to the expression's static type, but it can still fail at run time if the actual dynamic type is incompatible.

How to avoid it: Always do a second check for casts: ask whether the object's dynamic type can fulfill the cast type.

Confusing method-call compatibility with the receiver type

Pitfall: For `x.m(y)`, comparing `y`'s static type with `x`'s static type instead of with `m`'s parameter type.

Why it causes problems: The receiver `x` determines where the method is found, but the argument `y` must be checked against the method parameter.

How to avoid it: Split `x.m(y)` into two questions: is `m` available from `x`'s static type, and can `y`'s static type be assigned to `m`'s parameter?

Calling infinite recursion an infinite loop

Pitfall: Saying an incorrect recursive method has an infinite loop when the code has no loop construct.

Why it causes problems: The terminology hides the actual error: repeated method calls are filling the run-time stack.

How to avoid it: Use "infinite recursion" when the method repeatedly calls itself and never reaches a terminating base case.

Forgetting a base case

Pitfall: Writing a recursive method that always makes another recursive call.

Why it causes problems: Without a base case, there is no smallest problem that returns directly, so the recursion never terminates normally.

How to avoid it: Write the base case first and test the smallest inputs before writing the recursive case.

Not making the subproblem strictly smaller

Pitfall: Including a base case but calling the method with the same-sized problem, such as `factorial(n)` inside `factorial`.

Why it causes problems: The recursion may never move toward the base case, so the stack keeps growing.

How to avoid it: Before finalizing a recursive call, explicitly check that the argument represents a smaller problem, such as `n - 1`, a shorter string, or a smaller array range.

Using the stack trace as the main design tool

Pitfall: Trying to design a recursive solution by mentally simulating every call in the stack.

Why it causes problems: Stack tracing is precise but tedious, and it is not the best way to discover the recursive structure of a solution.

How to avoid it: Use a recursion tree and the divide-conquer-combine idea to design the solution; use the stack afterward to verify execution mechanics.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain the three inheritance reasoning layers Jackie emphasized: syntax, compile-time static-type checks, and run-time dynamic-type behaviour;

- trace dynamic binding for an overridden method by starting at the receiver's dynamic type and choosing the closest ancestor implementation;
- apply slide 90's type-checking rules to assignments, method calls, method-return assignments, casts, and method calls with cast arguments;
- distinguish compile-time cast validity from run-time cast safety;
- explain why `toString` follows the same overriding and dynamic-binding pattern as `equals`;
- describe what makes an array or return value polymorphic;
- explain what recursion is and distinguish direct recursion from other chains of method calls;
- identify base cases and recursive cases in a recursive definition;
- use the divide-conquer-combine pattern to describe a recursive solution;
- explain why recursive calls must be made on strictly smaller subproblems;
- translate the recursive factorial definition into Java code;
- trace `factorial(3)` using both a recursion tree and the run-time stack;
- identify why missing base cases and non-reducing recursive calls lead to stack overflow;
- state the recursive definition of Fibonacci numbers used in lecture and trace the structure of `fib(4)`;
- locate and use the posted Lab 4 solution, practice-test solution, CodingBat Recursion 1 exercises, past recursion lab, source code, and tutorial resources.

Suggested Practice

- Redo Lab 4 under time pressure, focusing on the inheritance-and-array pattern rather than memorizing the final code.
- Attempt the Programming Test 3 practice test once under a realistic time limit before watching or reading the solution walkthrough.
- Redraw the `Object -> A -> B -> C` hierarchy and create three cases: no override, override in `C`, and override in `B`; for each case, decide which `equals` version is called.
- Use the type-checking table by covering the condition column and reconstructing the rule for each code pattern.
- For each cast example you study, write two separate answers: why it compiles and whether it can throw `ClassCastException`.

- Study the `toString` slide and explain how it parallels the `equals` examples.
- Complete CodingBat Recursion 1 for the recursion style expected in this course.
- Reimplement `factorial` recursively, then deliberately write the two incorrect versions from lecture and explain why each fails.
- Trace `factorial(3)` twice: once with a recursion tree and once with the run-time stack.
- Complete Jackie's assigned `fib(4)` tracing exercise using both a recursion tree and a stack.
- After CodingBat Recursion 1, work through the posted past 2030 recursion lab and compare your solutions with the posted solutions.
- Treat CodingBat Recursion 2, Haskell practice, and Tower of Hanoi as optional enrichment after the required recursion practice is under control.

Where to Review

- **iPad/PDF notes, page 2:** administrative reminders, including the Lab 4 solution release and Programming Test 3 on July 27.
- **iPad/PDF notes, pages 3–5:** overridden methods and dynamic binding for `equals` in the `Object -> A -> B -> C` hierarchy.
- **iPad/PDF notes, page 6:** the summary table of type-checking rules and the prompt about when `ClassCastException` occurs.
- **iPad/PDF notes, page 7:** the general recursive problem-solving pattern: base cases, divide, recursive calls, and combination.
- **iPad/PDF notes, page 8:** run-time stack rules: push, pop, active call, suspended calls, and empty-stack termination.
- **iPad/PDF notes, pages 9–10:** factorial as a recursive definition, Java implementation, recursion-tree trace, and stack trace.
- **iPad/PDF notes, page 11:** comparison of stack tracing and recursion-tree tracing.
- **iPad/PDF notes, pages 12–13:** common recursion errors: no base case and non-smaller recursive calls.
- **iPad/PDF notes, pages 14–15:** Fibonacci definition and the `fib(4)` tracing exercise.
- **Lecture recording:** review the opening logistics if you need Programming Test 3 guidance, then focus on the dynamic-binding examples and the recursion introduction.
- **Posted course resources:** Lab 4 solution walkthrough, practice-test solution recording, CodingBat Recursion 1, the past recursion lab, posted Java source code, and the recursion tutorial section.

Closing Reminder

This digest gives you the study route through the lecture, not the full lecture. To use it well, return to Jackie's iPad/PDF notes and the recording, redraw the inheritance and recursion diagrams, re-trace the examples by hand, and write the Java code yourself. For Programming Test 3, do not only read the Lab 4 solution: redo the pattern under time pressure. For recursion, do not only read recursive code: practise defining base cases, making strictly smaller recursive calls, and tracing the calls with both a tree and a stack.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 23, July 28

Recursion on Strings, Recursion on Arrays: Efficient Subproblem Design

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 7

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 12

Main Ideas

- The lecture continued the recursion unit by moving from general recursion principles to concrete recursive methods on strings and the setup for recursion on arrays.
- Jackie reminded students that every recursive solution should be organized around base cases for small problems and recursive cases that create strictly smaller subproblems.
- The String methods needed for the exam in this context were reviewed: `isEmpty()`, `substring(...)`, `length()`, and `charAt(...)`.
- The technical detail behind `substring(i, j)` is essential: index `i` is included, but index `j` is excluded, so the included range is from `i` to `j - 1`.
- The palindrome example showed how to compare the first and last characters, then recursively check the middle substring only when that comparison succeeds.
- The string reversal example used a different pattern: remove the head of the string, recursively reverse the tail, then append the head at the end.
- The `occurrencesOf` string problem was left as an assigned exercise, using the same head-tail pattern with a numeric combination step.
- For arrays, Jackie contrasted a naive approach that creates new subarrays at every recursive call with the preferred approach of passing the same array reference plus index bounds.
- The efficient array-recursion pattern uses parameters such as `from` and `to` to describe the current subproblem range without copying array contents.
- For current preparation, Jackie emphasized Lab 5, final-exam readiness, recursion practice, and careful tracing of the provided examples.

Roadmap

1. Course logistics and immediate priorities

Jackie began by reminding students that only two lectures remained and that the remaining technical focus would be recursion examples and exercises. Lab 5 had been released late, so the original calendar deadline remained visible, but a grace period was available without penalty; students were told to check the Lab 5 PDF for the exact grace-period date.

Lab 5 was emphasized as a real study priority even though there would be no programming test on it. Its background topics, especially abstract classes and interfaces, may appear on the final exam and are meant to be practical takeaways after the inheritance unit. Jackie also noted that this week's office hour would be on Wednesday for 90 minutes rather than the usual Tuesday/Thursday pattern, that a review session may be arranged the following week, and that the final exam time and location had been fixed. Programming Test 3 grading was in progress, with an effort to release marks before the withdrawal deadline.

2. Quick recursion recap: base cases, recursive cases, and tracing

Before starting new examples, Jackie reviewed the divide-and-conquer pattern from the previous lecture. For a small problem, return a solution directly: that is a base case. For a larger problem, divide it into strictly smaller subproblems, make recursive calls, and combine the returned solutions into a solution for the original problem.

Jackie also reminded students to revisit the factorial and Fibonacci examples. In particular, `fib(4)` is worth tracing carefully because the recursive case has two recursive calls, so a stack trace requires more precision and patience than the factorial example. Students were directed to the posted tracing exercise and solution video.

3. String methods needed for recursive string problems

The technical work started with a Java `String` example using `s = "abcd"`. Jackie visualized a string as an indexed sequence of characters, similar to an array: indices `0`, `1`, `2`, and `3` contain `'a'`, `'b'`, `'c'`, and `'d'`.

The required String methods were reviewed carefully:

- `s.isEmpty()` checks whether the string has no characters.
- `s.substring(i, j)` returns the substring beginning at index `i` and ending before index `j`.
- `s.length()` returns the number of characters in a string.
- `s.charAt(i)` retrieves the character at index `i`.

Jackie explicitly contrasted `s.length()` for strings with `a.length` for arrays: the former is a method call with parentheses, while the latter is an array attribute.

4. The off-by-one idea behind `substring(i, j)`

A major part of the lecture was devoted to making `substring(i, j)` precise. The end index `j` is not included. Equivalently, the included characters are in the index range $[i, j-1]$, or in semi-open notation $[i, j)$. This detail is central for recursive string problems because the substring call often defines the strictly smaller subproblem.

For `s = "abcd"`, Jackie worked through representative substring calls: one returning the empty string, one returning the whole string, and one returning `"bc"`. Jackie also reviewed the interval-size formula: for an inclusive integer interval $[a, b]$, the size is $b - a + 1$.

5. Palindrome: compare the outside, recurse on the middle

The first recursive string problem was checking whether a string is a palindrome. Jackie used examples such as the empty string, a one-character string, `"madam"`, and `"racecar"` as palindromes, and examples such as `"man"` or strings with mismatched outer characters as non-palindromes.

The recursive idea is symmetric. If the string is empty or has length one, it is a palindrome immediately. Otherwise, compare the first and last characters. If they differ, the

whole string cannot be a palindrome. If they match, recursively check the middle substring. The middle substring is obtained with

`word.substring(1, word.length() - 1)`.

This call starts at index `1` and stops before `word.length() - 1`, so the last included index is `word.length() - 2`. Jackie emphasized this because using `word.length() - 2` as the second argument would be an off-by-one error.

6. Short-circuit evaluation in the palindrome method

The Java implementation combines two conditions with logical AND: the first and last characters must match, and the middle substring must recursively be a palindrome. Jackie connected this to short-circuit evaluation. If the left-hand side is false, Java does not evaluate the recursive call on the right-hand side. This avoids unnecessary work once the outer characters already prove that the word is not a palindrome.

The trace of `isPalindrome("abcba")` showed the recursion shrinking from `"abcba"` to `"bcb"` to `"c"`. The one-character string is the base case and returns `true`; those `true` results then propagate back through the earlier calls. Students should use this example to practise both the recursive decomposition and the return flow.

7. Reversal: reverse the tail, then append the head

The second string problem was returning the reverse of a string. Unlike palindrome checking, this method returns a new string rather than a boolean. Jackie used examples such as `"abc"` becoming `"cba"` and `"abcd"` becoming `"dcba"`.

The recursive solution splits the string into a head and a tail. The head is the first character, `s.charAt(0)`. The tail is everything after the first character, `s.substring(1, s.length())`. The recursive step is:

`reverseOf(tail) + head`.

This works because the first character of the original string should become the last character of the reversed string. Jackie traced `reverseOf("abc")` as `reverseOf("bc") + 'a'`, then `reverseOf("c") + 'b'`, and then the base case `"c"`, which builds back up to `"cba"`.

8. Another valid reversal strategy

Jackie also pointed out that recursive problems often have more than one correct strategy. The lecture's main reversal solution removed the head and appended it at the end after reversing the tail. As an exercise, Jackie suggested the opposite direction: remove the last character, recursively reverse everything before it, and put the last character at the front.

For example, for `"abcd"`, the last character `'d'` becomes the first character of the output, and the prefix `"abc"` should recursively become `"cba"`. Students should try to implement this alternative solution on paper or in Eclipse and compare its base cases and substring bounds with the lecture version.

9. Assigned string exercise: `occurrencesOf`

The PDF notes include an assigned exercise for `occurrencesOf(String s, char c)`. The method counts how many times a character occurs in a string. The base case is the empty string, which contributes zero occurrences. In the recursive case, the method separates the head character and the tail substring. If the head equals the target character, return $1+$ the recursive count of the tail; otherwise, return $0+$ the recursive count of the tail.

This exercise is important because it uses the same head-tail decomposition as reversal, but the combination step returns an integer rather than a string. It is a good bridge from simply tracing recursion to designing small recursive methods.

10. Naive array recursion: creating new subarrays

The lecture then shifted from strings to arrays. Jackie first showed a naive approach: when processing an array recursively, create a new array of length `a.length - 1`, copy all elements except the first into it, and recursively call the method on that new subarray.

The example with $\{A, B, C\}$ showed the pattern: the first recursive call creates $\{B, C\}$, and the next creates $\{C\}$. Although this approach can be made logically correct, Jackie emphasized that it is not the way to go. It wastes time and memory because every recursive call creates and fills another array object.

11. Efficient array recursion: same array reference plus index bounds

The preferred array-recursion pattern avoids new subarrays. Instead, pass the same array reference along with integer bounds, such as `from` and `to`. Conceptually, `from` and `to` describe the current subarray range inside the original array, but no new array object is created.

The notes show a helper form such as:

```
m(int[] a, int from, int to).
```

Typical base cases include `from > to` for an empty range and `from == to` for a one-element range. A recursive call such as `m(a, from + 1, to)` passes the same array reference `a` but shifts the starting index forward, thereby creating a strictly smaller conceptual subproblem. Jackie connected this to call by value: the array reference value is copied into the parameter, but the underlying array object is not copied.

Key Terms, Notation, and Structures

- **Base case**: A small input that can be solved directly without another recursive call, such as an empty string, a one-character string, or an empty array range.
- **Recursive case**: The part of a method that prepares one or more strictly smaller subproblems, makes recursive calls, and combines the returned results.
- **Strictly smaller subproblem**: A smaller version of the original input, such as a shorter substring or a shorter array interval; this is what lets recursion move toward a base case.

- `String.isEmpty()`: A String method that returns `true` exactly when the string has length zero.
- `String.length()`: A method call returning the number of characters in a string; do not confuse it with `array.length`.
- `array.length`: An array attribute, not a method call; it is written without parentheses.
- `String.charAt(i)`: A method that returns the character at index `i`, where valid indices range from `0` to `s.length() - 1`.
- `String.substring(i, j)`: A method that returns characters from index `i` through index `j - 1`; the second bound is exclusive.
- **Interval notation** $[i, j)$: A semi-open interval that includes i but excludes j; this matches the Java `substring(i, j)` convention.
- **Inclusive interval size**: For $[a, b]$ with both endpoints included, the size is $b - a + 1$, a fact Jackie used to reason about substring and subarray lengths.
- **Palindrome**: A string that reads the same forward and backward; recursively, the outer characters must match and the middle substring must also be a palindrome.
- **Short-circuit evaluation**: For `A && B`, Java skips `B` when `A` is already false; in the palindrome method, this can avoid an unnecessary recursive call.
- **Head**: The first character or first element of a sequence, such as `s.charAt(0)` for a string.
- **Tail**: The rest of the sequence after removing the head; for a string, this is produced by a substring starting at index `1`.
- `reverseOf`: A recursive string method whose lecture version computes the reverse of the tail and then appends the head.
- `occurrencesOf`: The assigned recursive string exercise that counts whether the head contributes 1 or 0, then adds the count from the tail.
- **Naive subarray recursion**: A recursion pattern that creates a new smaller array at each recursive step; Jackie presented it as correct in idea but inefficient in time and space.
- **Index-bounded array recursion**: The preferred array-recursion pattern, where `from` and `to` specify the current range within the same original array.
- **Call by value for array references**: Passing an array to a method copies the reference value into the parameter, not the whole array object.

Key Examples and Demonstrations

String method tester: `"abcd"`

Setup: The lecture used `s = "abcd"` and visualized it as an indexed sequence with indices `0` through `3`. Jackie evaluated calls involving `isEmpty()`, `substring`, `length()`, and `charAt`.

Main point: Recursive string methods depend on exact indexing. The most important rule is that `substring(i, j)` includes `i` but excludes `j`. A student who is off by one in the substring bounds will create the wrong subproblem.

What to practise: For `"abcd"`, recompute the three substring calls from lecture and the concatenation that splits the string at index 2. Then repeat with a different four- or five-character string.

Palindrome recursion: `isPalindrome`

Setup: The palindrome method returns `true` for empty and one-character strings. Otherwise, it compares the first and last characters and recursively checks the middle substring.

Main point: This example shows a symmetric recursive decomposition: discard both outer characters after verifying they match, then solve the smaller middle problem. It also shows why `word.substring(1, word.length() - 1)` is the right middle-substring call.

What to practise: Trace `isPalindrome("abcba")` and one non-palindrome of similar length. In each call, write the current string, the first-last comparison, the middle substring, and whether short-circuit evaluation stops or continues.

String reversal: `reverseOf`

Setup: The method returns the empty string for an empty input and returns the string itself for a one-character input. Otherwise, it separates the head from the tail, computes `reverseOf(tail)`, and concatenates the head at the end.

Main point: Unlike the palindrome example, the recursive result is not just a boolean to propagate. The result is built while the recursive calls return. For `"abc"`, the call waits for `reverseOf("bc")`, which waits for `reverseOf("c")`, and then the pieces are concatenated back into `"cba"`.

What to practise: Trace `reverseOf("abc")` and then `reverseOf("abcde")`. Mark the head, tail, recursive call, and final concatenation at each level.

Alternative reversal strategy

Setup: Jackie sketched a second approach: instead of moving the head to the end, move the last character to the front and recursively reverse the prefix before it.

Main point: Recursion is a design technique, not a fixed formula. The same problem may have multiple correct recursive decompositions, as long as the base cases are correct and the recursive call uses a strictly smaller input.

What to practise: Implement the alternative version for `"abcd"`: make `'d'` become the first output character and recursively reverse `"abc"`. Be especially careful with the substring bounds for the prefix.

Assigned exercise: `occurrencesOf`

Setup: The notes show `occurrencesOf(String s, char c)` with an empty-string base case, a head character, and a tail substring.

Main point: This example uses head-tail recursion with an integer return value. The combination step is $1+$ the tail count when the head matches the target, and $0+$ the tail count otherwise.

What to practise: Trace `occurrencesOf("abca", 'a')` and `occurrencesOf("abca", 'b')`. Then implement the method and compare your trace with actual output.

Naive array recursion by copying subarrays

Setup: The first array-recursion sketch creates a new array `sub` of length `a.length - 1`, copies `a[1]`, `a[2]`, and so on into `sub[0]`, `sub[1]`, and so on, then recursively calls `m(sub)`.

Main point: The conceptual subproblem is smaller, but the implementation is inefficient because every recursive call allocates a new array and copies elements.

What to practise: For $\{A, B, C\}$, draw the sequence of calls producing $\{B, C\}$ and then $\{C\}$. Count how many new arrays are created even in this tiny example.

Efficient array recursion by passing a range

Setup: The second array-recursion sketch uses `m(int[] a, int from, int to)`. The same array reference is passed at each call, while `from` and `to` define the current range.

Main point: The recursive call `m(a, from + 1, to)` shrinks the conceptual subarray by moving the left bound forward, but it does not copy the array. This is the standard pattern Jackie wants students to use for array recursion.

What to practise: For $\{A, B, C\}$, trace the ranges $[0, 2]$, $[1, 2]$, and $[2, 2]$. Say which base case each range eventually reaches and which elements are still under consideration.

Common Pitfalls and Misconceptions

Treating `substring(i, j)` as if `j` were included

Pitfall: Writing the second argument of `substring` as if it were the last included index.

Why it causes problems: This creates off-by-one errors. In the palindrome method, using `word.length() - 2` as the second argument would omit one more character than intended and check the wrong middle substring.

How to avoid it: Every time you see `substring(i, j)`, immediately translate it as "include `i` through `j - 1`". For the palindrome middle, the correct call is `word.substring(1, word.length() - 1)`.

Confusing `s.length()` with `a.length`

Pitfall: Writing `s.length` for a string or `a.length()` for an array.

Why it causes problems: These are different Java constructs. A string's length is obtained by a method call, while an array's length is an attribute. Mixing them causes compilation errors.

How to avoid it: Memorize the syntax pair: `s.length()` for strings, `a.length` for arrays.

Ignoring short-circuit evaluation in recursive boolean code

Pitfall: Thinking the recursive call in the palindrome method always runs.

Why it causes problems: This misses both the efficiency and the control flow of the method. When the first and last characters differ, the result is already false, so Java skips the recursive call on the middle substring.

How to avoid it: When tracing `A && B`, evaluate `A` first. Only evaluate `B` when `A` is true.

Creating new arrays in every array-recursive call

Pitfall: Implementing array recursion by allocating a new smaller array and copying elements at every recursive step.

Why it causes problems: The recursion may still be logically correct, but it wastes time and memory. Jackie explicitly described the repeated copying as not the way to go.

How to avoid it: Use a helper method with the original array reference plus index bounds, such as `from` and `to`. Shrink the bounds, not the array object.

Forgetting that recursive calls must shrink the problem

Pitfall: Making a recursive call that uses the same string, the same array range, or an otherwise non-smaller problem.

Why it causes problems: The recursion may never reach the base case. This leads to infinite recursion and eventually a stack overflow.

How to avoid it: Before writing the recursive call, say exactly what got smaller: the middle substring, the tail substring, the prefix substring, or the interval $[from, to]$.

Reading recursion examples passively

Pitfall: Looking at the completed code or trace and assuming the pattern has been learned.

Why it causes problems: Jackie emphasized that recursion requires practice. The intuition comes from working through examples, not from only reading finished solutions.

How to avoid it: Reproduce traces and implementations from scratch. Start with the lecture examples, then do CodingBat Recursion-1 and the provided practice materials.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain the divide-and-conquer structure of a recursive solution using base cases, recursive cases, smaller subproblems, and combination steps;
- identify the base cases and recursive case in the palindrome, reversal, and occurrence-counting examples;
- use `isEmpty()`, `length()`, `charAt(i)`, and `substring(i, j)` correctly in recursive string methods;
- translate `substring(i, j)` into the included index range $[i, j-1]$ and use that translation to avoid off-by-one errors;
- distinguish `s.length()` for strings from `a.length` for arrays;
- trace `isPalindrome("abcba")` and explain how short-circuit evaluation affects whether the recursive call is evaluated;
- implement and trace the head-tail version of `reverseOf`;
- design the alternative string-reversal solution that moves the last character to the front;
- implement and trace `occurrencesOf(String s, char c)` using the empty-string base case and head-tail recursive case;
- explain why creating new subarrays in recursive array code is inefficient;
- explain the preferred array-recursion helper pattern using the same array reference plus `from` and `to` index parameters;
- trace how a call such as `m(a, from + 1, to)` represents a strictly smaller conceptual subarray without allocating a new array;
- act on the course reminders: complete Lab 5 carefully, review its abstract-class and interface material for the final exam, and practise the provided recursion examples actively.

Suggested Practice

- Redo the String-method tester on paper for `"abcd"`: compute the outputs of `isEmpty()`, several `substring` calls, `length()`, and `charAt`.
- For at least five `substring(i, j)` calls of your own, write the included interval $[i, j-1]$, the length, and the resulting substring.
- Trace the palindrome method on `"abcba"` without looking at the notes, then trace a non-palindrome where short-circuit evaluation stops the recursion early.
- Implement `isPalindrome` in Eclipse using only the String methods Jackie listed, and test empty, one-character, palindrome, and non-palindrome cases.

- Trace `reverseOf("abc")` and `reverseOf("abcde")`; mark the head, tail, recursive return value, and final concatenation at each level.
- Implement the alternative reversal exercise where the last character is placed at the front and the prefix is reversed recursively.
- Complete the assigned `occurrencesOf` exercise from the notes, then test it on `"abca"` with target characters `'a'` and `'b'`.
- Draw the naive array-recursion trace for $\{A, B, C\}$, showing each new subarray created, then explain why this wastes time and space.
- Draw the efficient range-based array-recursion trace for $\{A, B, C\}$, using calls such as `m(a, 0, 2)`, `m(a, 1, 2)`, and `m(a, 2, 2)`.
- Practise CodingBat Recursion-1 problems after reviewing these examples; use the lecture patterns rather than guessing from memory.
- Complete Lab 5 properly and review abstract classes and interfaces as final-exam material.

Where to Review

- **Lecture recording, opening segment:** Review Jackie's reminders about Lab 5, Programming Test 3 grading, office hours, review-session planning, and final-exam logistics.
- **PDF/iPad notes, page 1:** Use the title page to situate the lecture: recursion on strings, recursion on arrays by creating subarrays, and recursion on arrays using call by value.
- **PDF/iPad notes, page 2:** Review the administrative reminders, especially the final exam time and Lab 5 priority.
- **PDF/iPad notes, pages 3–4:** Review the String-method examples, interval notation, and the size formula for inclusive intervals.
- **PDF/iPad notes, pages 5–6:** Revisit the palindrome design sketch and the Java implementation of `isPalindrome`, especially the middle substring and short-circuit `&&` logic.
- **PDF/iPad notes, pages 7–8:** Revisit the recursive string reversal method and the alternative reversal exercise.
- **PDF/iPad notes, pages 9–10:** Compare the naive subarray-copying approach with the preferred same-array-reference approach using `from` and `to`.
- **PDF/iPad notes, pages 11–12:** Complete the assigned `occurrencesOf` exercise and compare your reasoning against the displayed method structure.
- **Posted Java code:** Download and run the palindrome code Jackie mentioned; step through it in Eclipse if your trace does not match your expectation.

- **Posted recursion exercises:** Revisit the `fib(4)` tracing exercise and solution video from the previous recursion materials.
- **CodingBat Recursion-1:** Use it as the main outside practice set for exam-relevant recursion.

Closing Reminder

This digest is only a roadmap. To learn this lecture, you need to work with the recording, Jackie's iPad/PDF notes, the posted code, Lab 5, and the recursion exercises. Pay special attention to exact substring bounds, base cases, strictly smaller recursive calls, and the difference between copying arrays and passing an array reference with index bounds. Recursion will not become reliable just by reading completed examples; trace, code, test, and explain the examples yourself.

EECS2030 Advanced OOP (Summer'26)

Digest of Lecture 24, July 30

Recursion on Arrays: `allPositive`, `isSorted` and Course Wrap-Up

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 8

What You Should Be Able to Do 9

Suggested Practice 10

Where to Review 11

Closing Reminder 12

Main Ideas

- This final class completed the recursion unit by focusing on recursion over arrays using the `from/to` index-bound pattern.
- Jackie contrasted the inefficient strategy of creating new subarrays with the preferred strategy of passing the same array reference and changing only the bounds of the current interval.
- The generic array-recursion skeleton used two base cases: `from > to` for an empty interval and `from == to` for a one-element interval.
- The `allPositive` example showed how base-case return values should match the logic of the problem: "all elements have property P" is true for an empty collection because no counterexample exists.
- The `allPositive` and `isSorted` examples both used short-circuit `&&` so that recursion stops once the answer is already determined.
- The `isSorted` example clarified non-descending order, adjacent-pair checking, and the off-by-one danger of recursing with `from + 2` instead of `from + 1`.
- Jackie emphasized that tracing recursion, propagating return values, and possibly counting recursive calls are fair exam-style skills.
- The assigned recursion reading/exercise was Tower of Hanoi, including specification, legend, Java tracing, and optional running-time reasoning as a preview of later courses.
- The course wrap-up connected the lecture to larger review targets: exceptions, recursion, OOP, inheritance, polymorphism, dynamic binding, casting, `instanceof`, `equals`, debugging, and JUnit testing.

Roadmap

1. Final class and transition from strings to arrays

Jackie began by noting that this was the final class and that recursion on strings had already been covered in the previous lecture. The remaining string example, `occurrencesOf`, was assigned for self-study using the posted slides and video. Students were told to use the materials actively: study the input/output behavior, think recursively, and trace the Java code.

The main in-class goal was to finish the `from/to` index pattern for recursion on arrays. Jackie connected this to exam preparation by saying that array-recursion questions would use a similar skeleton and that students should practise with past lab material and CodingBat Recursion-1 style examples.

2. Why not create new subarrays?

The lecture resumed from the previous discussion of a naive array-recursion approach. In that approach, each recursive call creates a new array object of size `a.length - 1`, copies elements over, and calls the method on the copied subarray. Jackie described this as logically correct but very inefficient because it wastes both time and memory.

The better design is to pass the same array reference and use two integer parameters, `from` and `to`, to say which inclusive interval of the array is currently under consideration. The array object does not change; only the conceptual subproblem range changes.

3. The general `from/to` skeleton

Jackie reviewed the skeleton

```
m(int[] a, int from, int to)
```

where `from` and `to` describe the current inclusive interval $[\texttt{from}, \texttt{to}]$. The interval-size idea from the previous lecture reappeared: for an inclusive interval $[a, b]$, the size is $b - a + 1$. Thus $[3, 2]$ has size 0 and represents an empty interval, while $[3, 3]$ has size 1.

This leads to two standard base cases: `from > to` for an empty interval, and `from == to` for a one-element interval. A typical recursive call such as `m(a, from + 1, to)` shrinks the interval by moving the left boundary one step forward while passing the same array reference by value.

4. Tracing the skeleton on empty and non-empty arrays

The first trace used an empty array $a1 = \{\}$. The initial call `m(a1, 0, a1.length - 1)` becomes `m(a1, 0, -1)`, so `from > to` holds immediately and base case 1 applies.

The second trace used $a2 = \{A, B, C\}$. The initial range is $[0, 2]$, representing the whole array. The first recursive call moves to $[1, 2]$, then the second moves to $[2, 2]$, which reaches the one-element base case. This trace was meant to make clear that no new array is created; the subproblem is represented only by the changing index interval.

5. `allPositive`: problem meaning before code

The first concrete problem asked whether all integers in an array are positive. Jackie used inputs such as $\{\}$, $\{4\}$, $\{4, 7, 3, 9\}$, and $\{5, 3, -2, 9\}$ to clarify the expected return values before discussing code.

A key point was the empty-array case. For a universal statement such as "for every element x, property $P(x)$ holds," an empty collection makes the statement true because there is no counterexample. Jackie contrasted this with an existential statement such as "there exists an element satisfying P," which would be false for an empty collection because there is no witness.

6. `allPositive`: helper method and short-circuit tracing

The implementation separated the public method from the recursive helper. The public `allPositive(int[] a)` calls the helper with `0` and `a.length - 1`. The helper carries the recursive machinery with `from` and `to`; Jackie noted that the helper should be private in a good design.

For `allPositiveHelper`, the empty interval returns `true`, the one-element interval returns `a[from] > 0`, and the recursive case checks

```
a[from] > 0 && allPositiveHelper(a, from + 1, to).
```

The `&&` matters: if `a[from] > 0` is false, the recursive call is skipped. In the trace of $\{5, 3, -2, 9\}$, the checks at indices 0 and 1 continue, but the negative value at index 2 makes the left side false, so the later recursive call is bypassed and `false` propagates back to the original call.

7. Sorting orders and the meaning of non-descending

The second concrete problem asked whether an array is sorted in non-descending order. Before coding, Jackie clarified four sorting-order terms: descending/decreasing, ascending/increasing, non-descending, and non-ascending. The important distinction is that the word "non" introduces equality as an allowed case.

For non-descending order, Jackie derived the condition as "not descending." Since descending means an adjacent pair satisfies $a[i] > a[i+1]$, non-descending means $\neg(a[i] > a[i+1])$, which is $a[i] \leq a[i+1]$. The definition is local: every adjacent pair must satisfy the condition. A global-looking upward or flat trend is not enough if some adjacent pair violates the required relation.

8. `isSorted`: adjacent pairs and the `from + 1` warning

For `isSorted`, the base cases are again the empty interval and the one-element interval; both return `true` because no adjacent-pair violation can be found. In the recursive case, the method checks the current adjacent pair and then recurses:

```
a[from] <= a[from + 1] && isSortedHelper(a, from + 1, to).
```

Jackie emphasized that the recursive call must use `from + 1`, not `from + 2`. If the first comparison checks indices 0 and 1, the next recursive call must be able to check indices 1 and 2. Jumping to `from + 2` would skip an adjacent pair and could accept an unsorted array.

The trace of $\{3, 6, 5, 7\}$ showed this short-circuit behavior. The first pair $3 \leq 6$ succeeds, so the first recursive call is made. In that call, the pair $6 \leq 5$ fails, so the next recursive call is skipped. Jackie pointed out that counting how many recursive calls actually execute can be an exam-style question.

9. Assigned Tower of Hanoi material

The assigned recursion exercise was Tower of Hanoi. The notes include the strategy, a Java helper method, tracing pages, and a running-time recurrence page. Jackie directed students to the slides and posted video walkthrough. The running-time discussion was described as optional for this course, but useful as a preview of EECS2101-style analysis.

Students should treat Tower of Hanoi as practice for recognizing a recursive specification, identifying base cases, following helper parameters, and tracing nested recursive calls carefully.

10. Course wrap-up and exam-oriented review themes

Jackie ended by reviewing the major skills from the course. Procedural control flow included exceptions and exception tracing. Recursion was emphasized as a thinking pattern that will matter more in later data-structure and algorithm courses, especially with trees and graphs. OOP review included classes, attributes, methods, helpers, dot notation, context objects, aliasing, inheritance, static and dynamic types, substitution, polymorphism, dynamic binding, method parameters, polymorphic arrays and return types, type casting, `instanceof`, and `equals`.

Jackie also emphasized practical development habits: use an IDE debugger and write JUnit tests incrementally rather than coding everything first and hoping it works. Generics were mentioned as useful self-study for later courses, but not part of the final exam. Jackie also suggested longer-term self-study resources on algorithms and design patterns, and reminded students to review prerequisite math material before later courses.

Key Terms, Notation, and Structures

- **Array interval recursion**: A recursive array method represents the current subproblem using an inclusive index interval rather than by creating a new array.
- `from` **and** `to`: Integer parameters that specify the current interval $[\texttt{from}, \texttt{to}]$ of the array being considered.
- **Empty interval**: The case `from > to`; for example, $[0, -1]$ has size 0 and represents no elements.
- **One-element interval**: The case `from == to`; the interval contains exactly one array element.
- **Interval size**: For an inclusive integer interval $[a, b]$, the size is $b - a + 1$, when the interval is meaningful; this explains why $[3, 2]$ has size 0 and $[3, 3]$ has size 1.
- **Call by value for array references**: The array reference value is copied into the method parameter, so recursive calls can refer to the same array object without copying the array contents.

- **Public method and helper method**: The public method gives users a clean interface, such as `allPositive(a)`, while a helper method carries implementation details such as `from` and `to`; Jackie noted that the helper should be private.
- `allPositive`: A boolean array method that checks whether every element in an integer array is strictly greater than zero.
- **Universal quantification** $\forall$: A "for every" statement; on an empty collection it is true because there is no counterexample.
- **Existential quantification** $\exists$: A "there exists" statement; on an empty collection it is false because there is no witness.
- **Short-circuit** `&&`: Java evaluates the left side first; if it is false, the right side is not evaluated, so a recursive call on the right may be skipped.
- `isSorted`: In this lecture, a boolean method checking whether an integer array is sorted in non-descending order.
- **Non-descending order**: For every adjacent pair, $a[i] \leq a[i+1]$; equality is allowed.
- **Descending/decreasing order**: For every adjacent pair, $a[i] > a[i+1]$, using a strict downward relation.
- **Ascending/increasing order**: For every adjacent pair, $a[i] < a[i+1]$, using a strict upward relation.
- **Non-ascending order**: For every adjacent pair, $a[i] \geq a[i+1]$; equality is allowed.
- **Adjacent-pair checking**: The `isSorted` recursive step checks `a[from]` against `a[from + 1]`, then recurses from `from + 1` so the next pair is not skipped.
- **Tower of Hanoi**: The assigned recursion exercise involving moving disks among three pegs; the notes include strategy, Java code, tracing, and a recurrence-style running-time page.

Key Examples and Demonstrations

Same array reference with changing bounds

Setup: The generic method `m(int[] a, int from, int to)` was traced on an empty array and on $\{A, B, C\}$. The initial call uses `from = 0` and `to = a.length - 1`.

Main point: The same array reference is passed in each call. The interval changes from $[0, 2]$ to $[1, 2]$ to $[2, 2]$, so the subproblem shrinks without allocating new arrays.

What to practise: Redraw the trace for $\{A, B, C, D\}$. List the intervals reached by `m(a, from + 1, to)` and identify when each base case would apply.

`allPositive`: empty arrays and universal logic

Setup: The problem asks whether all numbers in an array are positive. Jackie discussed $\{\}$, $\{4\}$, $\{4, 7, 3, 9\}$, and $\{5, 3, -2, 9\}$.

Main point: The empty array returns `true` for `allPositive` because it represents a universal statement with no counterexample. This is a problem-specific base-case decision, not a random convention.

What to practise: Explain in words why `allPositive({})` returns `true`. Then contrast it with an `existsPositive` method, where the empty array should return `false`.

`allPositiveHelper`: short-circuit false propagation

Setup: The helper checks `a[from] > 0 && allPositiveHelper(a, from + 1, to)`. The trace used $\{5, 3, -2, 9\}$.

Main point: The first two positive values allow recursion to continue. At -2, the left side of `&&` is false, so the recursive call on the remaining range is skipped. The final result becomes `false` and propagates back through the previous calls.

What to practise: Trace $\{4, 7, 3, 9\}$, $\{5, 3, -2, 9\}$, $\{0\}$, and $\{-1, 2\}$. For each one, mark which calls actually execute and where short-circuiting occurs.

Sorting-order diagrams

Setup: Jackie used diagrams to distinguish descending, ascending, non-descending, and non-ascending arrays, with examples such as $\{5, 4, 3, 2, 1\}$, $\{1, 2, 3, 4, 5\}$, $\{1, 2, 2, 3, 4\}$, and $\{4, 3, 2, 2, 1\}$.

Main point: Sorting order is checked pairwise. Non-descending allows both increasing steps and flat steps, so the condition is $a[i] \leq a[i+1]$ for every adjacent pair.

What to practise: For each of the four sorting orders, write the adjacent-pair condition using $a[i]$ and $a[i+1]$. Then test the condition on two arrays with repeated values.

`isSorted`: why the recursive call uses `from + 1`

Setup: The non-descending `isSortedHelper` checks `a[from] <= a[from + 1]` and then calls `isSortedHelper(a, from + 1, to)`.

Main point: The recursive call must overlap with the previous comparison. After checking indices 0 and 1, the next call must check indices 1 and 2. Using `from + 2` skips a necessary adjacent pair.

What to practise: Trace $\{3, 6, 6, 7\}$ and $\{3, 6, 5, 7\}$. Count how many recursive helper calls are executed before the method returns.

Tower of Hanoi assigned exercise

Setup: The assigned notes show the Tower of Hanoi strategy for one, two, and three disks, then a Java helper using disk bounds and peg numbers.

Main point: Tower of Hanoi is an example where the recursive solution has two recursive calls around one direct move. It is useful for practising nested recursive structure

and for seeing a running-time recurrence, even though the running-time analysis is optional here.

What to practise: Follow the posted walkthrough for three disks $A < B < C$. Write the sequence of moves, then relate each move to the corresponding helper call in the trace.

Common Pitfalls and Misconceptions

Creating new subarrays at each recursive step

Pitfall: Building a smaller array object and copying elements before every recursive call.

Why it causes problems: The method may be logically correct, but it wastes time and memory. Jackie explicitly described this as inefficient and not the approach to use for array recursion.

How to avoid it: Pass the same array reference and represent the current subproblem with `from` and `to`. Change the bounds, not the array object.

Choosing base-case returns without thinking about the problem

Pitfall: Assuming every empty-array base case should return the same boolean value.

Why it causes problems: The correct value depends on the meaning of the method. For `allPositive`, the empty array returns `true` because a universal property has no counterexample. For an "exists" question, the empty collection would return `false` because there is no witness.

How to avoid it: Before writing code, state what the method means in words and decide what the empty interval should mean for that problem.

Forgetting that the recursive call must shrink the interval

Pitfall: Passing the same `from` and `to` values again, or continuing to check only index 0.

Why it causes problems: The recursion does not make progress toward a base case, or it repeatedly checks the wrong part of the array.

How to avoid it: In each recursive call, identify the new interval explicitly. For the examples in this lecture, the left bound moves from `from` to `from + 1`.

Ignoring short-circuit evaluation

Pitfall: Tracing as if the recursive call after `&&` always executes.

Why it causes problems: This gives the wrong execution trace and the wrong count of recursive calls. In both `allPositive` and `isSorted`, a false left-hand side prevents the right-hand recursive call from running.

How to avoid it: When tracing `A && B`, evaluate `A` first. Only draw or count `B` when `A` is true.

Using `from + 2` in `isSorted`

Pitfall: After checking `a[from]` and `a[from + 1]`, jumping to `from + 2` for the next recursive call.

Why it causes problems: This skips the adjacent pair `a[from + 1]` and `a[from + 2]`. An array such as $\{3, 6, 5, 7\}$ could be mishandled because the violation is between indices 1 and 2.

How to avoid it: Remember the intended sequence of checks: $(0, 1)$, $(1, 2)$, $(2, 3)$, and so on. Therefore the recursive call should use `from + 1`.

Confusing non-descending with a vague overall trend

Pitfall: Treating non-descending as a broad visual trend that can tolerate local decreases.

Why it causes problems: Sorting order is not judged by a loose big-picture curve. It is checked on every adjacent pair. One violating pair is enough for `isSorted` to return `false`.

How to avoid it: Translate non-descending into the precise condition $a[i] \leq a[i + 1]$ for every valid adjacent index i.

Leaving testing until after all code is written

Pitfall: Writing an entire method or programming-test solution without running tests until the end.

Why it causes problems: Bugs accumulate and become harder to isolate. Jackie used this as a practical warning when discussing responsible programming habits.

How to avoid it: Write small JUnit tests as you build the method. Include edge cases such as empty arrays, one-element arrays, equality cases, early false cases, and late false cases.

What You Should Be Able to Do

After this lecture, you should be able to:

- Explain why passing `from` and `to` is more efficient than creating new subarrays in recursive array methods.
- Identify the two standard base cases for inclusive array intervals: `from > to` and `from == to`.
- Trace the generic call sequence `m(a, 0, a.length - 1)` for empty arrays and small non-empty arrays.
- Explain why `allPositive({})` should return `true`, using the idea of universal statements and the absence of counterexamples.
- Contrast the empty-array return value for `allPositive` with an "exists positive" method.

- Implement `allPositive` with a public method and a private helper carrying `from` and `to`.
- Trace `allPositive` on arrays containing all positive values, zeros, and negative values, including where `&&` short-circuits.
- Define descending, ascending, non-descending, and non-ascending order using adjacent-pair inequalities.
- Derive non-descending order as $\neg(a[i] > a[i+1])$, which becomes $a[i] \leq a[i+1]$.
- Implement and trace `isSorted` for non-descending arrays using `a[from] <= a[from + 1]` and `from + 1`.
- Explain why `from + 2` is an off-by-one error in the recursive `isSorted` check.
- Count which recursive calls actually execute in examples where short-circuit evaluation stops the recursion early.
- Locate and use the assigned `occurrencesOf` and Tower of Hanoi materials for additional recursion practice.
- Review the course wrap-up topics that Jackie identified as important for final-exam preparation and future courses.

Suggested Practice

- Reproduce the generic range trace for $\{A, B, C\}$: start with $[0, 2]$, then trace the calls to $[1, 2]$ and $[2, 2]$. Repeat with four elements.
- Implement the public `allPositive` method with a private helper. Test $\{\}$, $\{4\}$, $\{4, 7, 3, 9\}$, $\{5, 3, -2, 9\}$, $\{0\}$, and $\{-1, 2\}$.
- For `allPositive`, draw a call tree or stack-style trace and mark which recursive calls are bypassed by short-circuit `&&`.
- Design an `existsPositive` method. Before coding, justify why the empty-array base case should return `false`.
- Write the adjacent-pair conditions for descending, ascending, non-descending, and non-ascending arrays. Then classify arrays with repeated values.
- Implement `isSorted` for non-descending order and test empty, one-element, already sorted, equality, early-violation, and late-violation cases.
- Modify the `isSorted` idea for non-ascending order by changing the comparison, not the whole recursive structure.
- Trace `isSorted({3,6,5,7})` and count the number of recursive helper calls that actually execute.

- Review the posted `occurrencesOf` string-recursion exercise and video if you did not already complete it after Lecture 23.
- Work through the Tower of Hanoi assigned slides and video. Focus on the specification, the helper-call structure, the trace for three disks, and optionally the recurrence for running time.
- For final-exam preparation, revisit exception tracing, type casts, `instanceof`, `equals`, polymorphism, dynamic binding, and JUnit testing habits.

Where to Review

- **iPad/PDF notes, page 1:** Lecture 24 overview: array recursion with `allPositive` and `isSorted`, Tower of Hanoi assigned readings, and course wrap-up.
- **iPad/PDF notes, page 2:** The same-array-reference pattern with `m(a, from, to)`, the empty and one-element interval base cases, and traces for $\{\}$ and $\{A, B, C\}$.
- **iPad/PDF notes, pages 3–8:** The `allPositive` problem, universal/existential reasoning for empty collections, the public/helper method structure, and traces including $\{5, 3, -2, 9\}$.
- **iPad/PDF notes, page 9:** Sorting-order diagrams and the distinction among descending, ascending, non-descending, and non-ascending order.
- **iPad/PDF notes, pages 10–15:** The `isSorted` problem, the correct `from + 1` recursive step, and traces for sorted and unsorted arrays.
- **iPad/PDF notes, pages 16–22:** Assigned Tower of Hanoi material, including strategy, Java helper, tracing, and optional running-time recurrence.
- **Lecture recording:** Review the traces for $\{5, 3, -2, 9\}$ with `allPositive` and $\{3, 6, 5, 7\}$ with `isSorted`. Focus on short-circuiting and return propagation.
- **Posted exercises:** Use the `occurrencesOf` string-recursion video and the Tower of Hanoi video if you want Jackie to walk through those examples with you.
- **Practice resources Jackie mentioned:** Past lab examples and CodingBat Recursion-1 style problems are appropriate practice for the recursive patterns in this lecture.
- **Course wrap-up topics:** Revisit exception tracing, recursion, array reasoning, inheritance, static and dynamic types, substitution, polymorphism, dynamic binding, casting, `instanceof`, `equals`, debugging, and JUnit tests. Jackie said generics are useful self-study, but not covered on the final exam.

Closing Reminder

This digest is a roadmap for the final lecture, not a substitute for the lecture recording, Jackie's iPad/PDF notes, posted exercises, labs, or your own coding and tracing practice. The most important work after this class is active: redraw the interval traces, implement the helpers, test edge cases with JUnit, explain the base-case logic, and practise short-circuit traces until you can do them without looking at the completed examples.