lectures # week-6

Week 6: Test 1 Review, Test 2 Preview & Group Project Introduction (Friday, February 13, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture introduces the Group Project and covers Sprint Zero requirements.

Lecture Preview [0:00]

Today's agenda:

1. **Test 1 review** — Debrief on grading, curve, and answer walkthrough
2. **Test 2 preview** — New question styles and practice problems
3. **Group project introduction** — Tetris project, teams, sprints, and expectations
4. **Sprint Zero overview** — Initial group meeting requirements

---

Admin Notes [0:30]

Test 1 Curve: A hard curve of 7-8 points was applied across the board. The curve would not take any student above 70/70.

- One natural 100 on the test (before the curve)
- After the curve, 3-4 students scored 100
- Class average after curve (excluding three zeros): **80%**

- Canvas class average appears lower because it includes zeros from one dropped student and two students who have effectively withdrawn

> **Grading Corrections**
>
> If something was marked wrong on your test that was discussed as correct in class, let the instructor know — mistakes happen and will be corrected. Also double-check that the score on the front page matches the individual scores inside, and verify the addition.

Test 1 Review [4:09]

Fill in the Blank & Matching [4:19]

These sections tested definitions. No widespread issues were noted. For the matching section, some students swapped the **transitive** and **symmetric** properties of equals.

Short Answer — Code Output [8:43]

BigDecimal immutability (Question 3):

```
BigDecimal x = new BigDecimal("5.0");
x.add(new BigDecimal("3"));
// Output: 5.0 (NOT 8.0)
```

`BigDecimal` is immutable — `x.add(...)` returns a **new** `BigDecimal` without modifying `x`. The result floats away and gets garbage collected.

Record equality (Question 4):

Two separate `Pair` objects (`p1` and `p2`) created with the same values:

- `p1 == p2` → `false` (different objects in memory)
- `p1.equals(p2)` → `true` (records auto-implement `equals` based on fields)
- `p3 = p1; p3 == p1` → `true` (same reference)

Key Concept

Java **records** automatically implement `equals()` and `hashCode()` based on their component fields. This is different from regular classes, which inherit `Object.equals()` (reference equality) by default.

Checkstyle Violations [10:19]

The question was worth 6 points on the page but only 4 points on the front — if you scored 4/6, you received full credit.

Key violations in the code:

- Instance field `rate` should be `myRate` (naming convention)
- All parameters need the `final` keyword
- Missing braces on `if` statement
- Missing Javadoc above class and method declarations
- Local variable `result` should be `final`
- Magic number `4` on line 8

Debugging Questions [12:03]

Question 1 — Overloaded vs. overridden `equals`:

```
// BUG: This OVERLOADS equals, not overrides
public boolean equals(FoodItem other) { ... }

// FIX: Parameter must be Object to override
public boolean equals(Object other) { ... }
```

The `@Override` annotation was intentionally removed — if present, the compiler would have caught this as an error. This is exactly why `@Override` protects you.

Question 2 — Missing assert:

Code did `new Product(null)` expecting a `NullPointerException`, but never called `assertThrows`. Without an assert statement, the test always passes regardless of what the code does.

Question 3 — Null check missing:

The test passed but contained a bug: the code didn't check if the object was `null` before calling `.isEmpty()`, which would throw a `NullPointerException` on null input. This question was given 4 free points due to an issue with the test case as written.

Question 4 — Broken symmetry in `equals`:

```
// BUG: StoreItem.equals uses instanceof AbstractItem
// This means a StoreBulkItem passes the instanceof check
// Result: a.equals(b) == true but b.equals(a) == false
```

The fix: use `instanceof StoreItem` (the concrete class) instead of `instanceof AbstractItem` (the parent). This prevents subclasses from breaking the symmetry contract of `equals`.

> ⚠ **Symmetry Rule**
>
> If `a.equals(b)` is `true`, then `b.equals(a)` **must** also be `true`. Using `instanceof` with a parent type in `equals` is a common way to accidentally break this contract.

Page 10-11 Questions [16:47]

Private field access in subclass: The subclass couldn't compile because it tried to directly access a `private` field in the parent class. Fix: call `super()` in the constructor to pass the value up to the parent.

Off-by-one in comparison: `qualifiesForDiscount` used `compareTo` incorrectly — needed `>= 0` (greater than or equal to the minimum) instead of just equals.

ArrayList reassignment bug: Every call to `addItem` created a new `ArrayList`, wiping out all previously added items. The `myItems = new ArrayList<>()` line inside `addItem` should not exist — the list should be initialized once (in the constructor) and only `add` to it afterward.

Implementation Questions [21:06]

- **`hashCode`:** Needed `Objects.hash(myName, myPrice)`
- **Static comparison method:** Return one item or the other based on `compareTo`
- **Cart total:** Required a loop to accumulate the total. Key mistake: `total.add(item)` instead of `total = total.add(item)` — remember `BigDecimal` is **immutable**, so you must reassign the result.

Test 2 Preview [22:54]

Test 2 is **next Friday, February 20th**. Practice problems are posted in the Discord announcements thread.

> ⚠ **Time Constraint**
>
> The instructor has a faculty presentation in the classroom at 12:30, so everyone must be finished and out by 12:25.

Question Style 1 — Pass/Fail Trace [23:41]

You will be given code (which may or may not contain a bug) along with several JUnit assert statements. Your job:

1. Read and trace the code
2. For each assert, determine: **pass** or **fail**?

Answers could be any combination — all pass, all fail, or a mix. There will be **multiple** questions in this style. Use the space provided to hand-trace the code.

> 🔥 **Study Strategy**
>
> Form a study group. Each person creates their own version of this question style, then swap and solve each other's problems. Building the question is studying in itself.

Question Style 2 — Implement from Tests [26:54]

You will be given JUnit tests that **define the specification** of a method (instead of Javadoc). Read and understand all the tests, then write the implementation.

- Some questions will use the **Road Rage API** — if you coded Road Rage, you have a head start
- Some questions may use the **Store API** (e.g., calculating cart totals)
- The `EnumSource` annotation may appear to parameterize tests across enum values

Preparation

If you haven't been doing the assignment work yourself, this will be significantly harder. Working through the assignments is direct preparation for this test style.

Question Style 3 — Lambda Expression Matching [31:00]

Given an interface, identify which lambda expression can be used with it (multiple choice, one correct answer).

Key steps:

1. Is it a **functional interface**? (exactly one abstract method — default methods don't count)
2. If not → no lambda can replace it (trick question, answer is "none")
3. If yes → match the lambda's **parameter types** and **return type** to the abstract method's signature

Example walkthrough: An interface with two methods — but one is a `default` method, so it IS a functional interface. The abstract method returns `String` and takes `BigDecimal`, so the lambda must have one `BigDecimal` parameter and return a `String`.

Question Style 4 — Write ActionListener Code [35:34]

Write a complete `addActionListener` call using **modern Java syntax** (lambda expressions, not anonymous inner classes).

Fill in the Blank [36:28]

Definition-style questions, same as Test 1. Example: "When the same method call produces different behavior depending on the actual object type, this is called **polymorphism**."

Group Project Introduction [45:24]

Overview [45:24]

The group project runs for the **final four weeks** of the quarter (Weeks 7-10). Groups are already assigned in Canvas under **People**.

- **34 students** on the roster → **8 groups**
- 6 groups of 4 students, 2 groups of 5
- Groups of 5 include a student who has effectively withdrawn — make every effort to reach out, but be realistic

Groups were assembled to **spread domain knowledge** evenly based on assignment and test performance.

The Project: Tetris [54:16]

You will be building **Tetris**:

- **Backend:** Provided as a JAR file (no access to source code). You work with the API documentation only.
- **Frontend:** You build the entire GUI
- **Final week:** Backend source code is provided, and you must add one extra feature to both frontend and backend

UI design is up to your group's creativity. Requirements include displaying: the game board, next piece (centered), current score, and movement controls — but layout and visual design is yours to decide.

Sprint Structure [55:09]

The project follows a simplified **Scrum** methodology with four one-week sprints:

Sprint	Week	Focus
Sprint 0	Week 7	Team building, project setup, git workflow
Sprint 1	Week 8	Active development (requirements released ~Friday of Week 7)
Sprint 2	Week 9	Active development
Sprint 3	Week 10	Active development + backend modifications

Sprint requirements are released approximately **Friday of the week before** the sprint starts.

Group Meetings with Instructor [1:12:08]

Lecture time will be taken during **Weeks 8, 9, and 10** (Monday of each week) for group meetings with the instructor:

- 8 groups, ~12 minutes each during the class period
- Sprints end Sunday night; meetings happen Monday
- Finals week will also have a group meeting slot (no traditional final exam)

Less lecture time means more responsibility on you to read the course guides independently.

Sprint Zero [59:40]

Sprint Zero runs all of next week (due **Sunday night after the test**). You are **not** actively developing Tetris yet.

Purpose: Build your team, set up your project, establish your workflow.

> ⚠ **Don't Start Early**
>
> Don't pressure your group to meet this weekend — everyone should be focused on Assignment 3 and Test 2 prep. Start Sprint Zero work on Monday (no class due to Presidents' Day).

Sprint Zero Requirements [59:47]

- Have your **initial group meeting** (must be synchronous — voice/video minimum, in-person preferred)
- Get the starter repo set up
- Complete a **git exercise**: everyone works on their own branch, merges together, resolves merge conflicts
- Build a simple **console UI** on top of the Tetris model to demonstrate you've practiced with the API

Initial Group Meeting Agenda [1:01:29]

The meeting follows a Google Doc template (access with your UW Gmail, not personal Gmail):

1. **Assign a meeting manager** — keeps the meeting on task and follows the agenda (not necessarily the group leader)
2. **Assign a scribe** — takes notes and documents decisions

> ⚠ **Be Mindful of Roles**
>
> Don't assign roles based on preconceived gender, nationality, or cultural expectations. Let people volunteer.

3. **Get to know each group member:**
   - Preferred name / what they want to be called
   - Where they took their 142/143 courses
   - Programming strengths and weaknesses — be honest
   - Time obligations (work, family, other classes)
   - Interests outside of school
4. **Discuss group structure** — decide on leadership model (one leader, democratic, rotating, etc.)
5. **Set regular meeting times** — leverage the gap between classes on Monday/Wednesday

On Being Honest About Abilities [1:06:25]

Not every CS student is a strong programmer — and that's okay. Computer Science is not "Computer Programming." If you're not a strong programmer:

- **Tell your group.** Let them know so roles can be assigned effectively.
- **Still contribute.** Testing, documentation, project management, and organization are all valuable.
- **Pair program.** Ask a stronger programmer to code together — they explain while you type. Both people learn.

> ⚠ **Don't Disappear**
>
> The worst thing you can do is take a task you can't complete and then go silent. If you're stuck, **communicate immediately**. Your group will help pick up the load. Groups fail when members stop talking.

Communication is Everything [49:38]

The single biggest predictor of group success is **communication**:

- Meet at least once per week (twice if possible), ideally in person
- Set up a **Discord server** with channels for your group — it's free and takes seconds
- If group dynamics aren't working after Sprint 1, **change your approach** for Sprint 2
- If problems persist, notify the instructor early — don't wait until it's too late

---

This lecture outline is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.