lectures # week-1

Week 1: Linting Tools and A1a Walkthrough (Wednesday, January 7, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture walks through Assignment A1a requirements.

Quick Updates [1:32]

GitHub/IntelliJ Authentication Issues:

- Option A (PAT): Personal Access Token with 90-day expiration—do this once and you're done for the quarter
- Option B (GitHub Desktop): Works but you'll need to use it for all assignments

JUnit 5 Setup Issue:

If you see red imports for `org.junit.jupiter.*`, your project isn't recognizing JUnit 5:

1. Right-click on the red import
2. Go to **More Actions → Add JUnit 5 to classpath**
3. Do NOT accidentally add JUnit 4 (different syntax, won't compile)

JUnit 4 vs JUnit 5

IntelliJ's default suggestion is often JUnit 4. We're using JUnit 5. If you accidentally added JUnit 4, go to Project Structure → Modules → Dependencies and remove it.

New Guide: Git and Version Control guide now available. Git and GitHub are NOT the same thing—read the guide.

What is a Linting Tool? [15:11]

The Compiler's Job

The Java compiler transforms your human-readable code into bytecode. During this process, it checks **syntax**—and syntax must be 100% correct.

Examples of syntax errors: - `nuM` vs `num` (case sensitivity) - Missing semicolons - Type mismatches - Undefined identifiers

```
int nuM = 5;
int result = num * 2;  // Compiler error: 'num' not found (case matters!)
```

Key insight from 142: Your enemy was the red squiggly line. As a software developer, those red lines are your *friend*—they catch problems before runtime.

Beyond Syntax: The Linting Problem

This code compiles perfectly:

```
public BigDecimal getPrice() {
    return null;  // Syntactically correct, but terrible
}
```

What happens when someone calls `getPrice()` ? **NullPointerException at runtime.**

The compiler doesn't catch this because it's syntactically correct. But it's almost certainly a bug waiting to happen.

> **Runtime Errors Ship to Users**
>
> Compiler errors prevent you from having a product. Runtime errors give you a broken product that reaches users. Which is worse? (Ask anyone who's crashed a $300 million Mars probe.)

The Sweater Analogy

Your code is like a sweater: - **Syntactically correct** = It covers you, keeps you warm, works as a sweater - **Has lint on it** = Doesn't look as sharp as it could be

A linting tool removes the lint from your syntactically correct code.

Two Types of Linting Tools [28:58]

1. IntelliJ Inspections

Built into IntelliJ. Looks for code that is syntactically correct but likely to cause: - Runtime exceptions - Logic errors - Bugs

Example warnings: - "Method returns null" - "Condition is always true" - "Variable might not be initialized"

2. Checkstyle

Enforces **coding style standards**. In the real world, teams agree on style conventions so all code looks consistent.

TCSS 305 Style Rules include:

- Curly braces on the **same line** (not next line)
- Instance fields must start with `my` prefix: `myRadius`, `myPrice`, `myName`
- Specific indentation, spacing, Javadoc requirements

```
// Checkstyle violation: instance field doesn't start with 'my'
private String badName;

// Correct:
private String myName;
```

> **Why Consistent Style Matters**
>
> You're in a dev shop of 35+ programmers. When reviewing code or debugging someone else's class, you want everything to look the same. The linting tool enforces this automatically.

Can you customize Checkstyle rules?

Absolutely. Should you? **Not for assignments**—you're coding to the company (TCSS 305) standards. For personal projects, go wild (use AI to help—editing Checkstyle XML by hand is painful).

Requirement 1: Verify Linting Tools [38:18]

1. Open the `WRONG` package
2. Open `CheckstyleRuleBreaker.java`
3. Verify you see Checkstyle warnings (yellow highlighting)
4. Delete the `WRONG` package

That's it. 20% of the assignment done by deleting something.

Requirement 2: Stop Using System.out.println [39:35]

The old way (142/143):

```
System.out.println("Debug: x = " + x);
```

The problem: `System.out.println` only goes to the console. Period. Full stop.

Why Use a Logger Instead?

A `Logger` provides:

1. **Log Levels** - Categorize messages as INFO, WARNING, SEVERE, etc.
2. **Filtering** - Show only warnings and above, or turn off logging entirely
3. **Multiple Destinations** - Console, file, database, HTTP API, or all of the above

```
// Logger setup
public static final Logger LOGGER =
Logger.getLogger(StarterApplication.class.getName());

// Usage
LOGGER.info(() -> "Application started");
LOGGER.warning(() -> "Something suspicious happened");
LOGGER.severe(() -> "Critical error!");
```

Controlling log output:

```
// Show all messages
LOGGER.setLevel(Level.ALL);

// Show nothing
LOGGER.setLevel(Level.OFF);

// Show only warnings and above
LOGGER.setLevel(Level.WARNING);
```

From Programmer to Software Developer

Replacing `System.out.println` with proper logging is one of the steps from being a programmer to being a software developer.

Requirement 3: Clean Up StarterApplication [49:15]

Class Members

A **member** of a class includes: - Methods (static and non-static) - Instance variables (non-static fields) - Static/class variables - Inner classes

Not members: Local variables (defined inside methods)

Utility Classes

A **utility class** has only static members. You never instantiate it.

Example: `Math` class

```
// You don't do this:
Math m = new Math();
m.sqrt(16);

// You do this:
Math.sqrt(16);
Math.pow(2, 8);
Math.floor(3.7);
```

The Default Constructor Problem

Does this class have a constructor?

```
public class StarterApplication {
    public static void main(String[] args) { }
}
```

Yes! The Java specification requires every class to have a constructor. If you don't write one, the compiler inserts a **default constructor**: - Visibility: `public` - Parameters: none

This means anyone can instantiate your utility class, which you never intended.

Preventing Instantiation

Step 1: Add a private constructor

```
private StarterApplication() {
    super();
}
```

Now only code inside the class can instantiate it.

Step 2: Make the class `final`

```
public final class StarterApplication {
    // ...
}
```

This prevents inheritance. Without `final`, someone could create a child class and instantiate *that*.

> **Software Development Mindset**
>
> "If we never want or need something to happen, do your best to make sure someone *can't* do it."

The final Keyword [1:06:39]

`final` is overloaded in Java (arguably poor language design):

Context	Meaning
Class	Cannot be inherited
Method	Cannot be overridden in child classes
Instance/class variable	Cannot be reassigned after initialization
Parameter	Cannot be reassigned inside the method
Local variable	Cannot be reassigned

The last three are semantically similar, but `final` on a class vs method vs variable mean different things.

Complete Utility Class Pattern

```
public final class StarterApplication {

    public static final Logger LOGGER =
        Logger.getLogger(StarterApplication.class.getName());

    static {
        LOGGER.setLevel(Level.OFF);
    }

    private StarterApplication() {
        super();
    }

    public static void main(final String[] theArgs) {
        LOGGER.info(() -> "Hello from the logger!");
    }
}
```

Key Takeaways

1. **Linters catch what compilers don't** - Syntax errors stop compilation; linting errors catch likely bugs
2. **Use loggers, not System.out.println** - Logging provides levels, filtering, and flexible output destinations
3. **Utility classes need protection** - Private constructor + `final` class prevents unintended instantiation
4. **The compiler helps you** - It inserts default constructors and super() calls; understand what it's doing
5. **final is overloaded** - Same keyword, different meanings depending on context

---

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