a1a # tooling

Linters and Code Quality

TCSS 305 Programming Practicum

This guide introduces linters and explains why they're essential tools for writing professional-quality code. You'll learn what linters catch that compilers miss, and how to use the specific tools configured for this course.

The Java Compiler: Your First Line of Defense

When you write Java code, the compiler (`javac`) is your first gatekeeper. It checks that your code follows Java's syntax rules and type system before creating bytecode that can run.

What the compiler catches:

```
// Syntax error - missing semicolon
int x = 5

// Type error - can't assign String to int
int count = "hello";

// Missing import
BigDecimal price = new BigDecimal("9.99");  // Won't compile without import

// Undefined variable
System.out.println(unknownVariable);
```

If any of these issues exist, your code won't compile. The compiler is strict and unforgiving—and that's a good thing.

What the compiler does NOT catch:

```
// Compiles fine, but is this good code?
public class thing {
    String n;
    double p;

    public thing(String x, double y) {
        n = x; p = y;
    }

    public void DoSomething() {
        // empty method
```

```
    }
}
```

This code compiles successfully. But look at it: - Class name `thing` doesn't follow naming conventions (should be `Thing`) - Field names `n` and `p` are meaningless - Parameter names `x` and `y` tell us nothing - Method `DoSomething` uses wrong casing (should be `doSomething`) - No documentation anywhere - Empty method body—is this intentional or forgotten?

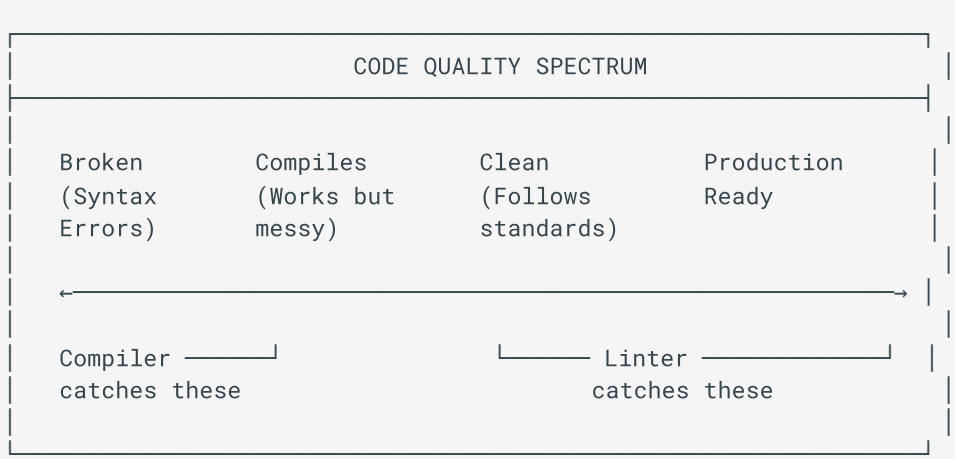
The compiler doesn't care about any of this. As long as the syntax is correct and types match, it's happy. This is where linters come in.

What is a Linter?

A **linter** is a static analysis tool that examines your code *without running it* and flags potential issues, style violations, and suspicious patterns.

> **Note**
>
> The term "lint" comes from the tiny fibers that collect on clothing. Just as a lint roller removes those small imperfections, a linter removes small code imperfections that accumulate over time.

Linters fill the gap between "code that compiles" and "code that's actually good."



Why Linters Matter

1. Consistency Across a Team

When multiple developers work on the same codebase, everyone has different preferences. One person uses 2-space indentation, another uses 4. One capitalizes constants, another doesn't. Without agreed-upon standards, the codebase becomes a patchwork of styles.

Linters enforce a single standard automatically. Everyone's code looks the same because the linter won't let inconsistent code through.

2. Catch Issues Early

Some patterns are technically valid but often indicate bugs:

```
// Empty catch block - swallows errors silently
try {
    riskyOperation();
} catch (Exception e) {
    // Nothing here... bug?
}

// Comparing strings with == instead of .equals()
if (name == "admin") {  // Almost certainly a bug
    grantAccess();
}

// Unused variable - did you forget to use it?
int total = calculateTotal();
return 0;  // total is never used
```

Linters flag these patterns before they become bugs in production.

3. Industry Standard

Every professional development team uses linters. Major companies like Google, Microsoft, and Amazon have extensive style guides enforced by automated tools. Learning to work with linters now prepares you for industry expectations.

> **Tip**
>
> Think of linter warnings like spell-check squiggles in a word processor. You *can* ignore them, but your document (or code) will be better if you address them.

Types of Issues Linters Catch

Formatting Issues

- Inconsistent indentation (tabs vs. spaces, 2 vs. 4 spaces)
- Line length exceeding limits (our limit: 120 characters)
- Missing or extra whitespace
- Brace placement style

Naming Convention Violations

- Class names not in `PascalCase`
- Method and variable names not in `camelCase`
- Constants not in `UPPER_SNAKE_CASE`
- In this course: instance fields must start with `my` prefix

Documentation Gaps

- Missing Javadoc on non-private classes and methods
- Missing `@param` or `@return` tags
- Missing `@author` and `@version` tags (required in TCSS 305)

Potential Bugs and Code Smells

- Empty blocks (catch, if, else, finally)
- Unused imports, variables, or parameters
- Magic numbers (unexplained numeric literals)
- Methods that are too long or too complex

Linters in the Wild

Different languages have different linting tools. Here are some you'll encounter:

Language	Popular Linters
Java	Checkstyle, SpotBugs, PMD, Error Prone
JavaScript	ESLint, JSHint
Python	Pylint, flake8, Black

Language	Popular Linters
C/C++	clang-tidy, cppcheck
Go	golint, staticcheck

> **Note**
>
> **SpotBugs** and **PMD** are Java tools that focus on finding potential bugs and code smells (deeper analysis than style checking). In TCSS 305, we use IntelliJ's built-in inspections instead of these tools—they provide similar functionality with better IDE integration.

Our Tools: Checkstyle + IntelliJ Inspections

In TCSS 305, we use two complementary tools:

Checkstyle

Checkstyle enforces coding style and documentation standards. It's configured with a rules file that defines exactly what's allowed.

Our configuration: `tcss305_checkstyle.xml` (Checkstyle 12.x)

Key rules we enforce: - Maximum line length: 120 characters - Instance fields must start with `my` prefix (e.g., `myName`, `myPrice`) - Javadoc required on all non-private types, methods, and fields - `@author` and `@version` tags required in class Javadoc - Lambda bodies limited to 1 statement (extract to helper methods) - Specific whitespace and brace placement rules

> **Guide**
>
> Checkstyle Rules Reference — Complete list of rules with examples.

IntelliJ Inspections

IntelliJ Inspections go deeper than style—they find potential bugs, performance issues, and code that could be simplified. This replaces tools like SpotBugs and PMD.

Examples of what inspections catch: - Unused variables or parameters - Possible null pointer dereferences - Redundant code that can be simplified - Complexity limits (cyclomatic

complexity, nesting depth, method count) - Class coupling and design issues

> **Guide**
>
> IntelliJ Inspections Reference — Complete list of enabled inspections and their limits.

> **Important**
>
> Before submitting any assignment, you must run both Checkstyle AND IntelliJ Inspections and address all issues. Code with warnings may lose points.

Example: Checkstyle-Compliant Code

Here's what well-formatted, Checkstyle-compliant code looks like:

```
/**
 * Represents an item available for purchase in the bookstore.
 * <p>
 * Each item has a name and a unit price. This class demonstrates
 * proper TCSS 305 coding conventions.
 *
 * @author Your Name
 * @version Winter 2025
 */
public class StoreItem implements Item {

    /**
     * The name of this item as displayed to customers.
     */
    private final String myName;

    /**
     * The unit price of this item in US dollars.
     */
    private final BigDecimal myPrice;

    /**
     * Constructs a StoreItem with the specified name and price.
     *
     * @param name  the name to assign to this item
     * @param price the unit price to assign to this item
     * @throws NullPointerException     if name or price is null
     * @throws IllegalArgumentException if name is empty or price is negative
     */
    public StoreItem(final String name, final BigDecimal price) {
        // validation and assignment here
        myName = name;
        myPrice = price;
```

```
    }
}
```

Notice: - **Javadoc with `@author` and `@version`** on the class - **`my` prefix** on instance fields: `myName`, `myPrice` - **Javadoc on non-private members** explaining purpose - **`@param` and `@throws` tags** documenting the constructor - **`final` keyword** on parameters (enforced by Checkstyle)

How-To: Running Checkstyle in IntelliJ

The Checkstyle-IDEA plugin is already configured in your project. Here's how to use it:

Check a Single File

1. Open the file you want to check
2. Right-click in the editor
3. Select **Check Current File** from the Checkstyle menu

Check the Entire Project

1. Open the **Checkstyle** tool window: **View → Tool Windows → Checkstyle**
2. Click the **Check Project** button (green play icon)
3. Results appear in the tool window below

Reading Results

Checkstyle results show: - **File and line number** where the issue occurs - **Description** of what rule was violated - **Severity** (error or warning)

Double-click any result to jump directly to that line in your code.

Fixing Violations

Most violations have straightforward fixes: - **Line too long:** Break into multiple lines - **Missing Javadoc:** Add documentation - **Wrong naming:** Rename the identifier - **Whitespace issues:** Add or remove spaces

> **Tip**
>
> Fix violations as you code rather than waiting until the end. It's much easier to fix one issue at a time than to face 50 violations at submission time.

How-To: Running IntelliJ Inspections

IntelliJ Inspections provide deeper code analysis beyond style checking.

Running Inspections

1. Go to **Analyze** → **Inspect Code...**
2. Choose scope:
3. **Whole project** - checks everything
4. **Current file** - checks just the open file
5. **Custom scope** - define your own
6. Click **OK**
7. Results appear in the **Inspection Results** tool window

Understanding Results

Results are grouped by category: - **Probable bugs** - issues likely to cause runtime problems - **Code style issues** - formatting and convention violations - **Declaration redundancy** - unused code - **Java language level migration aids** - suggestions for modern Java features

Fixing Issues

For many issues, IntelliJ offers quick fixes: 1. Click on an issue in the results 2. Press **Alt+Enter** (or click the lightbulb icon) 3. Select a suggested fix from the menu

> **Warning**
>
> Don't blindly apply all suggested fixes. Read each suggestion and make sure you understand what it's changing. Some "improvements" might not be appropriate for your specific situation. When in doubt, ask your professor!

Common Mistakes and How to Avoid Them

Ignoring Warnings Until Submission

Problem: You write all your code, then run the linter and find 47 violations.

Solution: Run Checkstyle frequently as you code. Configure IntelliJ to show violations in real-time (it does this by default with the plugin installed).

Suppressing Warnings Without Understanding

Problem: You add `@SuppressWarnings("all")` to make errors go away.

Solution: Understand why each warning exists. If you must suppress a warning, suppress only the specific one and add a comment explaining why.

```
// Acceptable - specific suppression with justification
@SuppressWarnings("MagicNumber")  // Screen dimensions are standard values
private static final int SCREEN_WIDTH = 1920;

// Not acceptable - blanket suppression
@SuppressWarnings("all")  // Don't do this!
```

Fighting the Style Guide

Problem: "But I prefer 4-space indentation!"

Solution: Consistency matters more than personal preference. In professional settings, you'll follow your team's style guide regardless of personal taste. This is good practice for that.

> **Gen AI & Learning: Linters and AI-Generated Code**
>
> AI coding assistants like GitHub Copilot and ChatGPT can generate code quickly, but that code often violates style guidelines and linter rules. When using AI tools:
>
> - **Always run Checkstyle and IntelliJ Inspections** on AI-generated code before committing
> - AI tools don't know our `my` prefix convention for instance fields
> - AI-generated Javadoc often lacks `@author`, `@version`, or proper `@param` descriptions
> - Treat AI output as a first draft that needs linting, not production-ready code
>
> Learning to fix linter violations helps you understand *why* conventions exist—knowledge that makes you a better developer than AI alone.

Summary

Tool	What It Checks	When to Run
Java Compiler (`javac`)	Syntax, types, imports	Every build (automatic)
Checkstyle	Style, naming, Javadoc, formatting	Frequently while coding
IntelliJ Inspections	Complexity, coupling, bugs, redundancy	Real-time in editor

Linters aren't obstacles—they're guardrails that help you write better code. The small effort of addressing warnings pays off in cleaner, more maintainable code that's easier for others (and future you) to read.

Further Reading

External Resources

- Checkstyle Official Documentation
- Google Java Style Guide - A widely-used industry standard
- IntelliJ IDEA Code Inspection - JetBrains documentation

References

Primary Texts:

- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. Chapter 1: Clean Code; Chapter 2: Meaningful Names.
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 56: Write doc comments for all exposed API elements.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 3: Fundamental Programming Structures.

Style Guides:

- Google Java Style Guide — Industry-standard style guide
- Oracle Code Conventions for Java — Original Sun/Oracle conventions

Linting Tools:

- Checkstyle Documentation — Official Checkstyle documentation
- Checkstyle Checks Reference — Complete rule reference
- IntelliJ IDEA Code Inspection — IDE inspection documentation
- Checkstyle-IDEA Plugin — IntelliJ plugin

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