a1a # java-fundamentals

Logging: Professional Output for Professional Code

TCSS 305 Programming Practicum

This guide explains why logging matters, how it differs from `System.out.println()`, and how to use Java's built-in logging framework effectively. You'll learn patterns you'll use throughout your career as a software developer.

The Problem with System.out.println()

You've probably used `System.out.println()` extensively in your programming courses. It's simple, it's familiar, and it works:

```
System.out.println("Starting calculation...");
System.out.println("x = " + x);
System.out.println("Done!");
```

For quick throwaway debugging while you're developing, this is fine. But for code you're going to submit or deploy, `System.out.println()` has serious limitations:

Why It Falls Short

Limitation	Problem
Output only to console	Can't redirect to files, monitoring systems, or dashboards
No way to turn off	Must delete or comment out every statement
No severity levels	Can't distinguish errors from informational messages
No timestamps	Hard to correlate messages with when things happened
No source information	Don't know which class produced the message

Limitation	Problem
Clutters production output	Users see debug messages meant for developers

Imagine releasing software to users and having debug messages appear everywhere. Or trying to diagnose a problem but having no timestamps to correlate with server logs. Or needing to add debugging back in because you deleted all your print statements.

So When IS System.out.println() Appropriate?

Use it for **quick throwaway debugging** that you'll delete within minutes:

```
// Temporary - delete before committing
System.out.println("DEBUG: value = " + value);
```

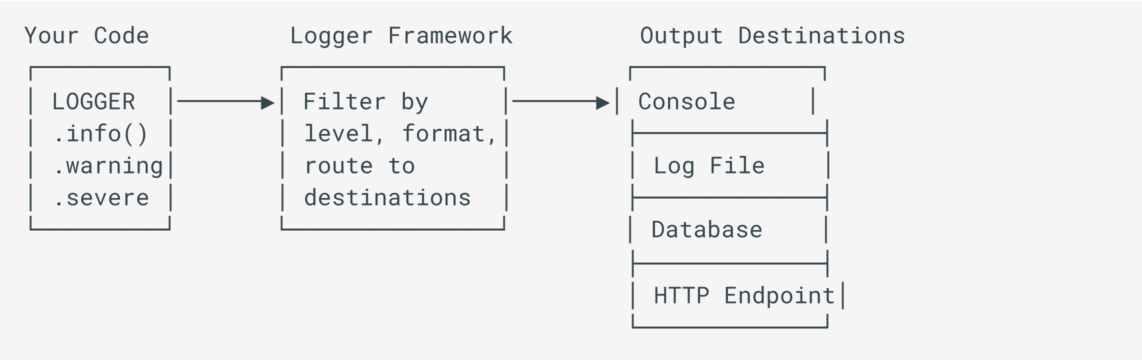
Even then, a Logger is often better because you can simply change the log level instead of deleting code.

Important

Course Rule: No `System.out.println()` or `System.err.println()` in submitted code. Checkstyle may flag it, and it will lose points. Use the Logger instead.

What is Logging?

Logging is a structured way to record application events. Instead of scattering print statements throughout your code, you send messages to a logging framework that decides what to do with them.



The key insight: **your code stays the same**, but you can change where messages go and which ones appear without touching the code. Flip a configuration switch to show debug

messages during development, then flip it back for production.

Logging Frameworks

Several logging frameworks exist in the Java ecosystem:

Framework	Description
java.util.logging	Built into Java, no dependencies needed
Log4j	Popular third-party framework (Apache)
SLF4J	Facade that works with multiple backends
Logback	Modern successor to Log4j

In TCSS 305, we use **java.util.logging** because it's built into Java. No external libraries to install, no dependency management to learn yet. The concepts transfer directly to other frameworks when you encounter them in industry.

Note

You'll likely encounter Log4j or SLF4J in internships and jobs. The core concepts (levels, loggers, handlers) are the same across all frameworks.

Log Levels: Severity Matters

Not all messages are equally important. A "Starting application" message is informational, but a "Database connection failed" message is critical. Log levels let you categorize messages by severity.

Java's Log Levels (java.util.logging)

From most severe to least severe:

Level	When to Use	Example
`SEVERE`	Something broke, application may not recover	`"Database connection failed"`
`WARNING`	Something concerning, but application continues	`"File not found, using defaults"`
`INFO`	General information about application state	`"Application started"`
`CONFIG`	Configuration information	`"Loaded 47 items from inventory"`
`FINE`	Debug information	`"Entering calculateTotal method"`
`FINER`	More detailed debug information	`"Loop iteration: i = 5"`
`FINEST`	Most detailed tracing	`"Variable x changed from 3 to 4"`

Setting the Level

You can control which messages appear by setting the logger's level:

```
// Show all messages (including FINE, FINER, FINEST)
LOGGER.setLevel(Level.ALL);

// Show only INFO and above (INFO, WARNING, SEVERE)
LOGGER.setLevel(Level.INFO);

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

When you set a level, you see messages **at that level and above**. Setting `INFO` means you see `INFO`, `WARNING`, and `SEVERE`, but not `CONFIG`, `FINE`, `FINER`, or `FINEST`.

Tip

During development, use `Level.ALL` to see everything. Before submitting, consider whether all those messages add value or just create noise.

Using Java's Built-in Logger

Getting a Logger Instance

Create a logger as a class constant:

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class MyClass {
    private static final Logger LOGGER =
Logger.getLogger(MyClass.class.getName());

    // ...
}
```

The `getLogger()` method takes a name (by convention, the fully-qualified class name). This name appears in log output, helping you identify which class produced each message.

Basic Usage

```
// Informational message
LOGGER.info("Application started successfully");

// Warning - something isn't quite right
LOGGER.warning("Configuration file not found, using defaults");

// Severe error - something is broken
LOGGER.severe("Failed to connect to database: " + ex.getMessage());

// Debug-level detail (only shows when level is FINE or lower)
LOGGER.fine("Processing item: " + item.getName());
```

String Concatenation and Lazy Evaluation

Be mindful of string concatenation in log messages:

```
// Eager evaluation - concatenation happens even if the message won't be
logged
LOGGER.fine("Processing item: " + expensiveOperation());
```

The problem? Even if `FINE` level logging is disabled, Java still evaluates the arguments—calling `expensiveOperation()` and building the string—before passing it to the logger. This wastes CPU cycles.

The solution: Lazy evaluation with lambdas

Java's Logger accepts a `Supplier<String>` (a lambda that returns a String). The lambda only executes if the message will actually be logged:

```
// Lazy evaluation - lambda only runs if FINE level is enabled
LOGGER.fine(() -> "Processing item: " + expensiveOperation());
```

What is lazy evaluation? Instead of computing a value immediately, you provide *instructions for how to compute it* (the lambda). The logger decides whether to follow those instructions based on the current log level. If the level is disabled, the lambda never runs—no string concatenation, no method calls, no wasted work.

For simple messages at INFO/WARNING/SEVERE (which are almost always enabled), regular concatenation is fine. For FINE/FINER/FINEST or messages involving expensive operations, prefer the lambda form.

Note

String literals are optimized by the compiler. Concatenating string literals like `"Hello " + "World"` happens at compile time, not runtime—the compiler combines them into a single `"Hello World"` string in the bytecode. The performance concern only applies to runtime concatenation involving variables or method calls, like `"Value: " + item.getPrice()`.

Two Patterns: Simple vs Industry Standard

There are two approaches to organizing loggers. Understanding both helps you recognize what you'll see in different codebases.

Simple Pattern (Used in Assignment 1A)

The starter code provides a centralized logger in `StarterApplication`:

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

    static {
        LOGGER.setLevel(Level.ALL);
    }
}
```

Any class can use this logger:

```java
public class StoreItem {
    public StoreItem(String name, BigDecimal price) {
        StarterApplication.LOGGER.info("Creating item: " + name);
        // ...
    }
}
```

Advantages: - One logger for the whole application - Easy to set up and understand - Good for learning the concepts

Limitations: - All messages appear to come from `StarterApplication` - Can't set different levels for different classes

Industry Standard Pattern (Future Assignments)

Each class has its own logger:

```java
public class StoreItem {
    private static final Logger LOGGER =
Logger.getLogger(StoreItem.class.getName());

    public StoreItem(String name, BigDecimal price) {
        LOGGER.fine("Creating StoreItem: " + name);
        // ...
    }
}

public class StoreCart {
    private static final Logger LOGGER =
Logger.getLogger(StoreCart.class.getName());

    public void add(ItemOrder order) {
        LOGGER.fine("Adding order to cart: " + order);
        // ...
    }
}
```

Advantages: - Log output shows which class produced each message - Can set different log levels per class - Better organization in large applications

Limitations: - More boilerplate (each class needs its own logger declaration)

Note

In Assignment 1A, we use the simple centralized pattern to focus on learning the concept. As you progress through the course and build larger applications, you'll transition to per-class loggers as used in industry.

When to Log

Good Times to Log

Situation	Level	Example
Application startup	`INFO`	`"Application started"`
Application shutdown	`INFO`	`"Shutting down gracefully"`
File I/O operations	`INFO` or `FINE`	`"Loaded 47 items from tacoma.txt"`
Configuration values	`CONFIG`	`"Using campus: Tacoma"`
Important state changes	`INFO`	`"Cart cleared"`
Errors and exceptions	`SEVERE` or `WARNING`	`"Failed to load inventory: " + ex.getMessage()`
Method entry/exit (debugging)	`FINE` or `FINER`	`"Entering calculateTotal"`

What NOT to Log

- **Every method entry/exit** - Too much noise, use only when debugging specific issues
- **Inside tight loops** - Performance impact and overwhelming output
- **Sensitive data** - Passwords, API keys, personal information
- **Obvious statements** - Don't log "About to add 2 + 2"
- **Things that clutter without adding value** - If it doesn't help diagnose problems, skip it

Caution

Never log passwords, API keys, credit card numbers, or personal information. Log files often end up in places others can access (servers, bug reports, support tickets). A leaked log file with credentials is a security incident.

Common Mistakes

1. Leaving System.out.println() in Submitted Code

Problem: You forget to remove debug print statements before submitting.

Solution: Use Logger from the start. Changing the log level is easier than hunting down print statements.

2. Forgetting to Remove Debug Logging

Problem: You submit with `LOGGER.setLevel(Level.ALL)` and your output is full of debug messages.

Solution: Before submitting, review your log level setting. `Level.INFO` is usually appropriate for submitted code.

3. Logging Sensitive Information

Problem: `LOGGER.info("User logged in with password: " + password);`

Solution: Never log credentials or personal data. Log the event, not the sensitive details: `LOGGER.info("User logged in: " + username);`

4. Over-Logging

Problem: So many log messages that the important ones get lost.

Solution: Be selective. Every log message should serve a purpose. Ask: "Would this help me diagnose a problem?"

5. Under-Logging

Problem: Something breaks and you have no information about what happened.

Solution: Log key events: startup, shutdown, file operations, errors, and significant state changes.

Gen AI & Learning: AI-Generated Code and Logging

When asking AI tools to generate Java code, they frequently include `System.out.println()` statements for debugging or output. Always review AI-generated code and replace any print statements with proper Logger calls before submitting. This is a good example of how AI tools often default to the "quick and easy" approach rather than professional best practices. Getting into the habit of mentally flagging `System.out` in AI suggestions helps you develop critical code review skills.

Logger Configuration in Our Project

The starter project configures the logger with a static initializer block:

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

    static {
        // Level.ALL - Display ALL logging messages
        // Level.OFF - Display NO logging messages
        LOGGER.setLevel(Level.ALL);
    }
}
```

The static block runs once when the class is first loaded, before any logger calls. This ensures the level is set before any messages are logged.

Changing the Level

To see fewer messages, change `Level.ALL` to another level:

```
static {
    LOGGER.setLevel(Level.INFO);  // Only INFO, WARNING, SEVERE
}
```

For submitted assignments, `Level.INFO` or `Level.WARNING` is typically appropriate unless the assignment instructions specify otherwise.

Summary

Concept	Key Point
Why not System.out	No levels, no timestamps, can't turn off, clutters output
What is logging	Structured event recording with configurable output
Log levels	SEVERE > WARNING > INFO > CONFIG > FINE > FINER > FINEST
Simple pattern	One centralized logger (used in 1A)
Industry pattern	Each class has its own logger
When to log	Startup, shutdown, I/O, errors, key state changes
What not to log	Passwords, every loop iteration, obvious statements

Logging is a fundamental skill for professional software development. The few minutes spent learning it now will save hours of debugging later and prepare you for industry expectations.

Further Reading

External Resources

- Oracle Java Logging Overview - Official Java documentation
- Baeldung: Java Logging Guide - Practical tutorial covering java.util.logging
- Java Logging Best Practices - Industry best practices
- SLF4J Manual - When you're ready to explore industry-standard logging facades

---

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 72: Favor the use of standard exceptions.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 7: Exceptions, Assertions, and Logging.

Language Documentation:

- Oracle JDK 25: java.util.logging — Official logging API documentation
- Oracle JDK 25: Logger — Logger class Javadoc
- Oracle JDK 25: Level — Log level documentation
- Oracle Java Logging Overview — Logging architecture guide

Industry Logging Frameworks:

- SLF4J Manual — Simple Logging Facade for Java
- Log4j 2 Documentation — Apache Log4j 2 (industry standard)

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