**Enum-Based Singleton**

Enum-Based singletons are attractive because they seem easy, light weight, and have several benefits.

```
public enum Logger {
    INSTANCE;

    public void log(String message) {
        System.out.println("LOG: " + message);
    }
}

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.INSTANCE;
        Logger logger2 = Logger.INSTANCE;

        logger1.log("Application started");
        logger2.log("Processing request");

        // Prove it is the same instance
        System.out.println(logger1 == logger2); // true
    }
}
```

In this example, Logger.INSTANCE is equivalent to Logger.getInstance(), which has some benefits.

- Java guarantees exactly one instance of each enum constant
- Enum initialization is thread-safe
- JVM prevents reflection attacks and multiple instances via serialization

So why not use them all the time?

Good for:

- Stateless or mostly stateless services
- Loggers
- Metrics collectors
- Feature flags
- App-wide utilities

Not ideal for:

- When you need lazy initialization with parameters
- When you want the pattern mechanics to be explicit
- When enum semantics may cause issues

Example:

```
public enum ConfigManager {
    INSTANCE;

    private final Path configPath;

    ConfigManager(Path path) { } // not allowed
}
```

In this example, you cannot pass a file path, credentials, select environments, or defer configuration of the controller. Sometimes you may see add an init method to get around it. The init method would bring in configuration parameters.

```
Logger.INSTANCE.init(config);
```

This is known as two-phase initialization, but introduces issues:

- Ordering (remember the JVM can reorder steps)
- Race conditions (multiple threads trying to initial it simultaneous)
- Invalid states (from multiple threads manipulating it)

Also note:

- Enums are already created when the Enum class is loaded, not first used, so there is an initial cost to it. If startup time matters, creation is expensive, or the singleton might not be used, then you might pay a high cost.
- Remember that Enums are concrete, final, cannot be subclassed, cannot be replaced, and hard to mock with testing suites (e.g., unit testing, Mockito).
- Enums can have state, but you can accidentally create global mutable state, implicit coupling, and order-dependent bugs (like mentioned above).

**Summary of Some Scenarios**

| **Scenario** | **Enum Singleton** |
|---|---|
| Logger/Metrics | Good |
| Simple stateless utility | Good |
| Configuration with parameters | Not Good |
| Lazy initialization is important | Not Good |
| Unit testing | Not Good |
| Multiple implementations | Not Good |

**Rule of Thumb:** Use enum singletons for simple, infrastructure-level services with no configuration and no lifecycle complexity. Everything else, use holder style.