Java Coding Standards

IGNITE BRILLIANCE

ARRAY

Contents

JAVA Coding Guidelines (v1.0)

Standard principles:

- **Keep It Simple (KISS)**
    - Write clean, clear, and understandable code. Avoid overengineering and unnecessary complexity.
- **DRY (Don't Repeat Yourself)**
    - Avoid code duplication by creating reusable methods, classes, and components.
- **Write Tests**
    - Ensure correctness with unit tests, and practice Test-Driven Development (TDD) for better reliability and maintainability.
- **Handle Errors Gracefully**
    - Properly catch, log, and handle exceptions to prevent the application from crashing and to ensure smooth error recovery.
- **Containerization**
    - Use containerization (e.g., Docker) to package your Java applications and their dependencies into portable, consistent environments. This improves deployment, scaling, and portability across different environments (development, testing, production).

1. Standardize the use of google format code

- Follow the set up here and ensure to enable it for all the projects: google/google-java-format: Reformats Java source code to comply with Google Java Style.

2. Naming Conventions

Proper naming conventions ensure that code is easy to understand. Java follows certain naming conventions for classes, methods, variables, constants, and other elements:

- **Classes and Interfaces**:
    - Use **CamelCase** with the first letter capitalized.
    - Class names should be nouns or noun phrases.
    - Example: EmployeeManager, OrderProcessor.

- **Methods**:
    - Use **camelCase** (first letter lowercase) for method names.
    - Method names should generally be verbs to indicate an action.
    - Example: calculateTotal(), findEmployeeById().
- **Variables**:
    - Use **camelCase** for variables.
    - Use descriptive names that reflect the purpose of the variable.
    - Example: totalAmount, userName.
- **Constants**:
    - Use **UPPER_CASE** with underscores separating words.
    - Constants are usually declared as static final.
    - Example: MAX_RETRY_COUNT, DEFAULT_TIMEOUT.
- **Packages**:
    - Use lowercase letters for package names, and use reverse domain naming.
    - Example: com.company.projectname.
- **Parameters**:
    - Use descriptive names in method parameters.

- Example: int customerId, String fileName.

3. Commenting and Documentation

Comments and documentation are essential for explaining the purpose and logic of the code.

- **Javadoc**:
  - Use Javadoc comments (/** */) for documenting classes, methods, and parameters.
  - Provide meaningful descriptions, especially for public methods, classes, and complex logic.
- **Inline Comments:**
  - Use inline comments to explain why specific code is used when it's not immediately obvious.
  - Place comments above complex or non-obvious code.
  - Keep comments concise and to the point.

4. Code Efficiency

- **Avoid Repetitive Code**:
  - Use methods to encapsulate repeated logic. Apply the DRY (Don't Repeat Yourself) principle.
- **Prefer Immutability**:
  - Where possible, make your objects immutable. Immutable objects are thread-safe and easier to reason about.

5. Braces and Line Breaks:

- Always use braces {} for blocks of code even for single statements. This avoids confusion when adding new code later.

6. Design Patterns

- Apply Design Patterns Appropriately:

- Familiarize yourself with common design patterns (e.g., Singleton, Factory, Observer, Strategy, etc.). They can help solve recurring design problems efficiently and improve code maintainability.

7. Testing

- **Write Unit Tests:**
    - Write unit tests for your code. Use frameworks like JUnit and Mockito.
    - Ensure the coverage is above 75% and you are testing the business logic across positive and negative test cases

8. Logging Best Practices

- **Use Logging Frameworks:**
    - Instead of using System.out.println() for logging, use proper logging frameworks such as SLF4J, Logback, or Log4j2. This allows for better control over log levels, log formatting, and output destinations (e.g., file, console).
- **Log Important Events:**
    - Log important events such as method entry and exit, exceptions, and key business operations.
- **Avoid Logging Sensitive Information:**
    - Never log sensitive information like passwords, personal data, or API keys in production.
- **Use Appropriate Log Levels:**
    - Use log levels (e.g., DEBUG, INFO, WARN, ERROR) appropriately to provide meaningful insights during debugging and production.
- The below format is preferred but this may change based on requirement

logback.xml

9. Version Control and Git

- **Commit Often, with Clear Messages:**

- Make regular commits with clear, concise messages. Follow the convention (e.g., Fix bug in payment logic or Add unit tests for user service).
- Keep JIRA detailed and link your commit messages to the JIRA ID you are working on.

- **Avoid Large Commits:**
  - Avoid making huge commits with too many changes. Instead, break your changes into smaller, more focused commits.

10. Performance Considerations

- **Minimize Object Creation:**
  - Reuse objects where possible. Avoid creating unnecessary objects in performance-critical areas of your code (e.g., within loops).
- **Use Object Pooling:**
  - For objects that are expensive to create (e.g., database connections), consider using an object pool (e.g., Apache Commons Pool) to reuse instances rather than creating new ones every time.
- **Use StringBuilder for String Concatenation:**
  - When concatenating strings in loops or in large blocks of code, use StringBuilder instead of + to avoid creating intermediate immutable String objects.
- **Use final for Performance:**
  - Mark variables and methods as final wherever possible. This helps the JVM optimize them during compilation (e.g., inlining).
- **Avoid Synchronization Overhead:**
  - Synchronization can be expensive in terms of performance. Avoid synchronizing non-critical code blocks or prefer alternatives like ConcurrentHashMap or AtomicInteger.

12. Security Best Practices

- **Use Strong Hashing for Passwords:**

    - Never store passwords in plain text. Use secure algorithms like bcrypt, PBKDF2, or Argon2 for hashing passwords before storage.

- **Validate User Input:**
    - Always validate and sanitize user input to prevent SQL injection, XSS (Cross-Site Scripting), and other security vulnerabilities.
- **Use Parameterized Queries:**
    - Use prepared statements or ORM frameworks like Hibernate to prevent SQL injection when interacting with databases.
- **Use HTTPS for Communication:**
    - Always use HTTPS (SSL/TLS) for secure communication, especially when transmitting sensitive data like login credentials or personal information.