**Java Coding Specification**

1. Naming Conventions

1.1 Classes and Interfaces

Use PascalCase (e.g., CustomerAccount, DatabaseService)

Use nouns for class names (e.g., Employee, RequestHandler)

Interfaces may use adjectives (e.g., Runnable, Serializable)

1.2 Methods

Use camelCase (e.g., calculateSalary(), getUserInfo())

Use verbs for method names (e.g., performCalculation(), validateInput())

1.3 Variables

Use camelCase (e.g., customerName, accountBalance)

Avoid single-letter names except in loops

Use meaningful, descriptive names

1.4 Constants

Use UPPER\_SNAKE\_CASE (e.g., MAX\_CONNECTIONS, DEFAULT\_TIMEOUT)

1.5 Packages

Use lowercase (e.g., com.company.project.module)

Use reverse domain name as prefix

2. Code Structure

2.1 File Organization

One class per file (except inner classes)

Member order:

Class variables

Instance variables

Constructors

Methods

Group related methods together

2.2 Indentation and Braces

Use 4 spaces for indentation (no tabs)

Use K&R style braces:

java

public void method() {

// code

}

2.3 Line Length

Maximum 120 characters per line

Break long lines at logical points

3. Documentation

3.1 JavaDoc

Document all public classes, methods, and fields

Use @param, @return, and @throws where appropriate

/\*\*

\* Calculates the area of a circle.

\* @param radius the radius of the circle

\* @return the calculated area

\* @throws IllegalArgumentException if radius is negative

\*/

public double calculateArea(double radius) {

// implementation

}

3.2 Inline Comments

Use for complex logic explanations

Avoid stating the obvious

4. Programming Practices

4.1 Exception Handling

Never use empty catch blocks

Catch specific exceptions rather than generic Exception

Use try-with-resources for resource management

try (FileInputStream input = new FileInputStream("file.txt")) {

// use resource

} catch (IOException e) {

// handle exception

}

4.2 Null Safety

Use Optional for return values that might be null

Validate method parameters using Objects.requireNonNull()

Avoid returning null from methods

4.3 Immutability

Prefer immutable objects where possible

Use final for fields that shouldn't change

Consider using records for data carriers (Java 14+)

4.4 Generics

Use generics for type safety

Avoid raw types

5. Modern Java Features

5.1 Language Features

Use var for local variables when type is obvious

Prefer pattern matching for instanceof checks (Java 16+)

Use text blocks for multi-line strings (Java 15+)

5.2 API Usage

Prefer Streams API for collection processing

Use new Date/Time API (java.time) instead of legacy Date/Calendar

6. Testing

6.1 Test Structure

Follow Arrange-Act-Assert pattern

Use descriptive test method names

One assertion per test method (when practical)

6.2 Test Naming

Use methodName\_scenario\_expectedResult pattern

@Test

void calculateTax\_belowThreshold\_returnsZero() {

// test implementation

}

7. Build and Dependencies

7.1 Dependency Management

Use Maven or Gradle for dependency management

Keep dependencies updated and minimal

Use specific versions rather than version ranges

7.2 Modularization

Consider using Java Platform Module System (JPMS) for large applications

Maintain clear module boundaries

8. Security

8.1 Input Validation

Validate all external inputs

Use prepared statements to prevent SQL injection

Sanitize data for XSS prevention in web applications

8.2 Sensitive Data

Never log sensitive information

Use secure storage for passwords and secrets

Consider using dedicated security libraries

Example Code

package com.example.project.service;

import java.util.Optional;

/\*\*

\* Service for user management operations.

\*/

public final class UserService {

private static final int MAX\_LOGIN\_ATTEMPTS = 3;

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = Objects.requireNonNull(userRepository);

}

/\*\*

\* Authenticates a user with provided credentials.

\* @param username the username to authenticate

\* @param password the password to verify

\* @return Optional containing user if authentication succeeds

\* @throws AuthenticationException if authentication fails

\*/

public Optional<User> authenticate(String username, String password) {

Objects.requireNonNull(username);

Objects.requireNonNull(password);

return userRepository.findByUsername(username)

.filter(user -> validatePassword(user, password));

}

private boolean validatePassword(User user, String inputPassword) {

// Password validation logic

return true; // Simplified for example

}

}

**Code Quality Tools**

Use static analysis tools (Checkstyle, PMD, SpotBugs)

Configure IDE to enforce coding standards

Use SonarQube for code quality monitoring

Set up pre-commit hooks for basic checks