## Java 技术管理规范

本规范旨在为 Java 项目开发提供统一的技术指导和标准，以提高代码质量、开发效率和系统稳定性。本规范参考了Google、阿里巴巴、美团等大型互联网公司的最佳实践，并结合实际项目经验进行总结。

### a、强制 (Must)

以下规则为 Java 项目开发必须遵守的规范，任何违反都将被视为严重问题。

1. **命名规范：**
   * 【强制】所有包名都应使用小写字母，多级包名之间用点隔开，且包名应具备业务领域含义。
     + **正例：** com.example.project.user.service, org.apache.commons.lang3
     + **反例：** com.example.Project.User.Service, com.Example.project
   * 【强制】类名、接口名、枚举名和注解名都应使用大驼峰命名法（PascalCase）。
     + **正例：** UserService, DataLoader, OrderStatusEnum, RequestMapping
     + **反例：** user\_service, dataLoader, Orderstatusenum
   * 【强制】方法名、变量名、参数名都应使用小驼峰命名法（camelCase）。
     + **正例：** getUserInfo(), userName, pageSize
     + **反例：** get\_user\_info(), user\_name, pagesize
   * 【强制】常量名都应使用全大写字母，并用下划线连接单词。
     + **正例：** MAX\_PAGE\_SIZE, DEFAULT\_TIMEOUT
     + **反例：** maxPageSize, DefaultTimeout
   * 【强制】抽象类命名应以 Abstract 或 Base开头。
     + **正例：** AbstractController, BaseService
     + **反例：** ControllerAbstract, ServiceBase
   * 【强制】异常类命名应以 Exception结尾。
     + **正例：** UserNotFoundException, IOException
     + **反例：** UserNotFoundErr, IOError
   * 【强制】测试类命名应以被测试的类名加 Test结尾。
     + **正例：** UserServiceTest, OrderServiceTest
     + **反例：** TestUserService, OrderService
   * 【强制】领域模型中的POJO（Plain Old Java Object）类，禁止出现任何业务逻辑，只包含属性和其对应的 getter/setter 方法。
     + 正例：
     + public class User {   
        private Long id;   
        private String name;   
        // getter and setter methods   
       }
     + 反例：
     + public class User {   
        private Long id;   
        private String name;   
        // getter and setter methods   
        public void processOrder(Order order) { // 包含业务逻辑   
        // ...   
        }   
       }
2. **代码格式：**
   * 【强制】使用 4 个空格作为缩进，禁止使用 Tab 字符。
     + 正例：
     + public void method() {   
        int i = 0; // 4个空格缩进   
       }
     + 反例：
     + public void method() {   
        int i = 0; // Tab 缩进   
       }
   * 【强制】
   * 大括号 ({})的使用遵循 K&R 风格，即左大括号不换行，右大括号换行。
     + 正例：
     + if (condition) {   
        // code   
       } else {   
        // other code   
       }
     + 反例：
     + if (condition)   
       {   
        // code   
       }
   * 【强制】方法参数之间、操作符之间、逗号之后应有一个空格。
     + **正例：** int a = b + c;, method(arg1, arg2);
     + **反例：** int a=b+c;, method(arg1,arg2);
   * 【强制】
   * 单行代码长度不超过 120 字符，超出应换行，并保持良好的可读性。
     + 正例：
     + String longVariableName = "This is a very long string that needs to be broken into multiple lines " +   
        "to adhere to the line length limit.";
   * 【强制】避免多余的空行，方法之间、类成员之间只留一个空行。
     + 正例：
     + public class MyClass {   
        private int field1;   
        private String field2;   
          
        public void method1() {   
        // ...   
        }   
          
        public void method2() {   
        // ...   
        }   
       }
     + 反例：
     + public class MyClass {   
        private int field1;   
          
          
        private String field2;   
          
          
        public void method1() {   
        // ...   
        }   
          
          
        public void method2() {   
        // ...   
        }   
       }
   * 【强制】类中成员变量定义应遵循：公共静态常量、私有静态常量、私有成员变量的顺序，并且属性声明前应有空行分隔。
     + 正例：
     + public class MyClass {   
        public static final String PUBLIC\_CONST = "public";   
          
        private static final String PRIVATE\_CONST = "private";   
          
        private String instanceVar;   
       }
3. **代码结构与设计：**
   * 【强制】禁止捕获了异常但不进行任何处理，或只是简单地打印堆栈信息。
     + 正例：
     + try {   
        // operation   
       } catch (IOException e) {   
        log.error("File operation failed: {}", e.getMessage(), e); // 记录日志并处理   
        throw new CustomBusinessException("File process error", e);   
       }
     + 反例：
     + try {   
        // operation   
       } catch (Exception e) {   
        e.printStackTrace(); // 不做任何处理   
       }
   * 【强制】禁止在循环体内创建对象，除非是业务逻辑必须且对象生命周期短暂，避免内存抖动。
     + 正例：
     + List<String> results = new ArrayList<>();   
       for (int i = 0; i < 100; i++) {   
        String s = String.valueOf(i); // 字符串是不可变对象，每次循环都会创建新对象，但这是合理的。   
        results.add(s);   
       }
     + 反例：
     + for (int i = 0; i < 100000; i++) {   
        User user = new User("name" + i); // 大量创建对象，可能导致频繁GC   
        // ...   
       }
   * 【强制】禁止直接使用 ThreadLocal，除非明确知道其生命周期并正确处理，防止内存泄漏。
     + **正例：** 使用InheritableThreadLocal或者结合线程池使用，并在使用后调用remove()清除。
     + **反例：** public static ThreadLocal<String> currentUser = new ThreadLocal<>(); 且不清理。
   * 【强制】禁止使用 Magic Number（魔法数字），应定义为有意义的常量。
     + **正例：** public static final int MAX\_USERS = 100; 然后使用 MAX\_USERS
     + **反例：** if (count > 100) { ... } (这里的 100 就是魔法数字)
   * 【强制】禁止使用原始类型作为集合的泛型参数（例如 List），应指定具体的泛型类型。
     + **正例：** List<String> names = new ArrayList<>();
     + **反例：** List names = new ArrayList();
   * 【强制】禁止在代码中硬编码敏感信息（如数据库密码、API Key 等），应使用配置中心或环境变量。
     + **正例：** 从配置文件中读取 String dbPassword =config.getProperty("db.password");
     + **反例：** String dbPassword = "password123";
   * 【强制】避免过长的方法，一个方法最好只做一件事，长度通常不超过 50 行，超过应考虑拆分。
     + 正例：
     + public void processOrder(Order order) {   
        validateOrder(order);   
        calculatePrice(order);   
        saveOrder(order);   
        sendNotification(order);   
       }   
       // 每个方法负责一个独立的子任务   
       private void validateOrder(Order order) { /\* ... \*/ }   
       private void calculatePrice(Order order) { /\* ... \*/ }   
       // ...
   * 【强制】避免过长的类，一个类通常不应超过 500 行，若功能过多应考虑拆分。
     + **正例：** 拆分为 UserService, UserDao, UserController 等。
     + **反例：** 一个 UserManage 类包含了所有用户相关的逻辑，从DB操作到接口定义。
4. **异常处理：**
   * 【强制】业务异常必须使用自定义异常类，并携带明确的错误码和错误信息。
     + **正例：** throw new UserNotFoundException(ErrorCode.USER\_NOT\_FOUND, "用户不存在");
     + **反例：** throw new RuntimeException("用户不存在");
   * 【强制】禁止在 finally 块中抛出异常，这会覆盖 try 或 catch 块中抛出的异常。
     + 正例：
     + try {   
        // ...   
       } finally {   
        if (resource != null) {   
        try {   
        resource.close();   
        } catch (IOException e) {   
        log.error("Failed to close resource", e); // 记录日志，不抛出   
        }   
        }   
       }
   * 【强制】禁止在 catch 块中忽略异常，至少需要记录日志。
     + 正例：
     + try {   
        // ...   
       } catch (Exception e) {   
        log.error("An error occurred: {}", e.getMessage(), e);   
       }
     + 反例：
     + try {   
        // ...   
       } catch (Exception e) {   
        // Do nothing   
       }
5. **注释规范：**
   * 【强制】
   * 类、接口、方法、枚举和常量必须包含 Javadoc 注释，说明其功能、参数、返回值、抛出的异常、作者、创建日期等。
     + 正例：
     + /\*\*   
        \* 用户服务接口。   
        \* 负责用户相关的业务逻辑操作。   
        \* @author yourname   
        \* @since 2023-01-01   
        \*/   
       public interface UserService {   
        /\*\*   
        \* 根据用户ID获取用户信息。   
        \* @param userId 用户ID   
        \* @return 用户对象，如果不存在则返回null   
        \* @throws UserNotFoundException 如果用户不存在   
        \*/   
        User getUserById(Long userId) throws UserNotFoundException;   
       }
   * 【强制】避免无意义的注释，如
   * // todo: fixme
   * ，应改为具有实际意义的描述或在任务管理系统中记录。
     + **正例：** // FIXME: Need to add pagination logic here, see JIRA-1234
     + **反例：** // TODO
   * 【强制】修改代码时必须更新相关注释，保持注释与代码一致。
     + **正例：** 修改方法签名后，同步更新Javadoc中参数和返回值的描述。
6. **安全规范：**
   * 【强制】禁止直接拼接 SQL 语句，必须使用预编译语句（PreparedStatement）防止 SQL 注入。
     + **正例：** PreparedStatement ps = conn.prepareStatement("SELECT \* FROM users WHERE name = ?"); ps.setString(1, userName);
     + **反例：** Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery("SELECT \* FROM users WHERE name = '" + userName + "'");
   * 【强制】避免将敏感数据（如密码、身份证号）明文存储或传输，应进行加密处理。
     + **正例：** 存储密码时使用加盐哈希，传输时使用HTTPS。
   * 【强制】对用户输入进行严格的校验和过滤，防止 XSS、CSRF 等攻击。
     + **正例：** 使用 OWASP ESAPI 或 Spring Security 等库进行输入过滤。

### b、推荐 (Recommended)

以下规则为 Java 项目开发中强烈推荐遵守的规范，有助于提升代码质量和可维护性。

1. **命名规范：**
   * 【推荐】POJO 类中避免出现任何业务逻辑，保持其纯粹性。
     + **正例：** 同强制规范。
   * 【推荐】枚举命名应体现其业务含义，并以 Enum结尾。
     + **正例：** OrderStatusEnum, UserRoleEnum
     + **反例：** OrderStatus, UserRole (可能与类名混淆)
   * 【推荐】接口的实现类命名应以Impl结尾。
     + **正例：** UserServiceImpl, OrderDaoImpl
     + **反例：** UserService, OrderDao (可能与接口名混淆)
   * 【推荐】通用工具类命名以Utils或Util结尾。
     + **正例：** DateUtils, StringUtils
     + **反例：** DateHelper, StringProcessor
2. **代码格式：**
   * 【推荐】使用 IDE 的代码格式化功能，并统一团队的代码格式配置。
     + **正例：** 团队成员都使用 IntelliJ IDEA，并导入相同的代码风格XML文件。
3. **代码结构与设计：**
   * 【推荐】使用 Lombok 简化 POJO 代码，但需注意其注解可能带来的兼容性问题。
     + 正例：
     + @Data // @Getter, @Setter, @ToString, @EqualsAndHashCode, @RequiredArgsConstructor   
       public class UserDto {   
        private Long id;   
        private String name;   
       }
     + **反例：** 手动编写大量的 getter/setter。
   * 【推荐】尽量使用 Java 8 及以上版本的新特性，如 Stream API、Lambda 表达式、Optional，以简化代码并提高可读性。
     + **正例：** List<String> names = users.stream().map(User::getName).collect(Collectors.toList());
     + **反例：** 使用传统 for 循环遍历集合并手动添加。
   * 【推荐】尽可能使用 Builder 模式创建复杂对象，尤其当对象有多个可选参数时。
     + **正例：** User user = User.builder().id(1L).name("Test").email("test@example.com").build();
   * 【推荐】尽量使用 Optional 来处理可能为空的对象，避免空指针异常，使代码更健壮。
     + **正例：** Optional.ofNullable(user).map(User::getName).orElse("Unknown");
     + **反例：** String name = (user != null) ? user.getName() : "Unknown";
   * 【推荐】对于集合类，推荐使用接口而非具体实现类来声明变量。
     + **正例：** List<String> names = new ArrayList<>();
     + **反例：** ArrayList<String> names = new ArrayList<>();
   * 【推荐】避免在循环中进行数据库查询或远程调用，考虑批量操作或数据预加载，减少IO次数。
     + **正例：** 批量查询 List<User> users = userDao.getUsersByIds(idList);
     + **反例：** 在循环中 for (Long id : idList) { User user =userDao.getUserById(id); }
   * 【推荐】使用 try-with-resources 语句管理资源（如输入输出流、数据库连接），确保资源被正确关闭。
     + 正例：
     + try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {   
        String line = reader.readLine();   
       } catch (IOException e) {   
        // handle exception   
       }
     + **反例：** 手动在 finally 块中关闭资源。
   * 【推荐】使用枚举代替常量类来表示有限的固定值，提供更好的类型安全性和可读性。
     + **正例：** public enum OrderStatus { PENDING, PROCESSING, COMPLETED }
     + **反例：** public class OrderStatusConstant { public static final int PENDING = 1; }
   * 【推荐】服务层（Service）方法参数建议使用数据传输对象（DTO），而不是多个散列参数，方便扩展和管理。
     + **正例：** userService.createUser(CreateUserRequest request);
     + **反例：** userService.createUser(String userName, String password, String email);
   * 【推荐】避免使用大量的 If-Else 嵌套，可以考虑使用多态、策略模式、状态模式等设计模式优化。
     + 正例：
     + interface PaymentProcessor { void process(Order order); }   
       class AlipayProcessor implements PaymentProcessor { /\* ... \*/ }   
       class WechatPayProcessor implements PaymentProcessor { /\* ... \*/ }   
       // 根据支付类型选择对应的处理器   
       PaymentProcessor processor = PaymentProcessorFactory.getProcessor(paymentType);   
       processor.process(order);
     + 反例：
     + if (paymentType.equals("Alipay")) {   
        // Alipay logic   
       } else if (paymentType.equals("WechatPay")) {   
        // WechatPay logic   
       }
4. **异常处理：**
   * 【推荐】对于业务异常，应该捕获并转换为统一的业务异常码返回给调用方，而不是直接抛出底层异常。
     + **正例：** catch (SQLException e) { throw new BusinessException(ErrorCode.DB\_ERROR, "数据库操作失败", e); }
   * 【推荐】在异常信息中包含足够的信息，如错误发生的时间、具体异常类型、相关参数等，方便排查问题。
     + **正例：** log.error("Failed to create user. User ID: {}, Error: {}", userId, e.getMessage(), e);
5. **性能优化：**
   * 【推荐】在 String 拼接时，优先考虑使用 StringBuilder 或 StringBuffer，尤其是在循环中。
     + **正例：** StringBuilder sb = new StringBuilder(); for (String s : list) { sb.append(s); }
     + **反例：** String result = ""; for (String s : list) { result += s; }
   * 【推荐】对于性能敏感的场景，考虑使用原始类型而非包装类型，避免自动装箱/拆箱带来的性能损耗。
     + **正例：** int count = 0;
     + **反例：** Integer count = 0; (在大量计算时)
   * 【推荐】合理使用缓存，减少重复计算和数据库查询。
     + **正例：** 使用 Guava Cache 或 Redis 缓存常用数据。
6. **日志规范：**
   * 【推荐】使用 SLF4J + Logback/Log4j2 等成熟的日志框架，禁止直接使用
   * System.out.println
     + **正例：** private static final Logger log = LoggerFactory.getLogger(MyClass.class); log.info("User created: {}", user.getName());
     + **反例：** System.out.println("User created: " + user.getName());
   * 【推荐】合理设置日志级别（TRACE, DEBUG, INFO, WARN, ERROR），生产环境避免输出过多的 DEBUG 信息。
     + **正例：** log.debug("Debug message, only for development");
     + **反例：** 生产环境日志充斥大量 DEBUG 信息。
   * 【推荐】
   * 日志中包含关键的业务上下文信息，方便问题追踪。
     + **正例：** log.error("Order processing failed for order ID: {}, reason: {}", orderId, e.getMessage());

### c、允许 (Allowed)

以下规则为 Java 项目开发中允许存在的规范，但需谨慎使用，并确保不会对项目造成负面影响，通常需要经过团队讨论和一致同意。

1. **命名规范：**
   * 【允许】在极少数情况下，为了简化代码，局部变量可以适当简化命名，但仍需保持可读性。
     + **正例：** 在短小的循环中，for (int i = 0; i < n; i++)，i是允许的。
     + **反例：** List l = new ArrayList(); (命名过于随意，不推荐)
2. **代码格式：**
   * 【允许】在某些特定场景下（如链式调用、DSL 定义），允许适当增加单行代码长度，但仍需保持可读性，且不得超过团队约定的最大长度。
     + **正例：** obj.method1().method2().method3().method4();
   * 【允许】在特殊情况下，为了对齐代码或提升可读性，可以进行额外的空行或空格调整
     + 正例：
     + String name = "Alice";   
       int age = 30;   
       String city = "New York";
     + (这里为了对齐等号，增加空格，虽然不严格符合空格规范，但提高可读性。)
3. **代码结构与设计：**
   * 【允许】在性能要求极高的场景下，且经过充分评估和测试，允许直接使用
   * new Thread()
   * 创建线程，但必须自行管理线程生命周期，包括线程池化，避免资源耗尽。
     + **正例：** new Thread(() -> { /\* custom logic \*/ }).start(); (但更推荐使用线程池)
   * 【允许】在进行少量数据操作或非关键业务逻辑时，允许不使用事务，但需确保数据一致性，并考虑潜在的风险。
     + **正例：** 对仅影响独立非关键数据的操作，如记录日志到数据库。
   * 【允许】为了方便调试，允许在开发环境中使用简单的日志输出（例如
   * System.out.println();
   * 或低级别的日志），但在生产环境必须替换为专业的日志框架，并调整日志级别。
     + **正例：** 开发阶段 System.out.println("Debug info: " + value);
   * 【允许】在非常特定的、性能瓶颈极端的场景下，且经过严密测试，允许使用反射或字节码操作来达到目的，但应避免滥用，因为它可能导致代码难以理解和维护。
     + **正例：** 使用反射调用私有方法进行单元测试。
4. **异常处理：**
   * 【允许】对于一些非关键性的、不影响核心业务流程的异常，可以考虑在日志中记录而不进行业务中断，但需要评估风险，并确保不会因此导致系统出现更严重的问题。
     + **正例：** catch (NotificationException e) { log.warn("Failed to send notification: {}", e.getMessage()); } (邮件发送失败不影响主业务)。
5. **第三方库使用：**
   * 【允许】为了快速验证功能或进行原型开发，允许引入非核心的第三方库，但需在后期评估其稳定性、性能、兼容性以及是否存在安全漏洞，并在正式发布前决定是否替换或移除。
     + **正例：** 在原型阶段引入一个小众的图形库来快速展示功能。