**Clean Code Coding Guidelines :: Java Spring Boot API**

**1. Naming Conventions**

* **Classes/Interfaces:** Use **meaningful names** that clearly indicate the purpose of the class. The name should be a **noun** for classes and interfaces, and typically use **CamelCase**.
* Example: OrderService, ProductController, CustomerRepository.
* **Methods:** Methods should be named with a **verb** to indicate an action, and the name should describe the action clearly. They should have names that are starting with lowercase
* Example: createOrder(), calculateTotalPrice(), sendEmailNotification().
* **Variables:** Use **descriptive names** that provide context. Avoid abbreviations unless widely accepted (e.g., orderAmount, productList).
* Example: orderTotal, userList, responseMessage.
* **Constants:** Use **UPPER\_CASE** with underscores to separate words.
* Example: MAX\_RETRIES, DEFAULT\_TIMEOUT.

**2. Code Structure and Organization**

Another best practice for Java clean coding is to avoid creating unnecessary objects. It is known as one of the best memory-consuming operations in Java.

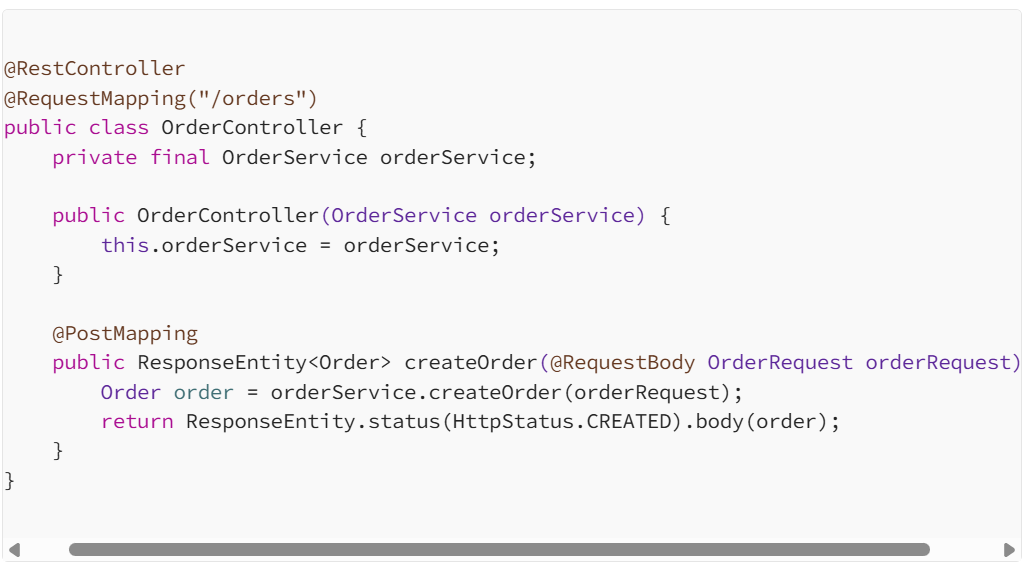
* **Package Naming:** Use a consistent and clear package structure. Organize by **domain** or **feature** rather than technical layers.
* Example: com.example.store.orders, com.example.store.payment, com.example.store.inventory.
* **Avoid Long Methods:** **Methods should be small** and do one thing. If a method is over 30–40 lines, it’s a sign that it should be broken up into smaller methods.
* Example: If processOrder() has logic for validation, calculation, and sending notifications, refactor it into separate methods like validateOrder(), calculateOrderTotal(), and sendConfirmationEmail().
* **Classes Should Be Focused:** Each class should have a **single responsibility**. Avoid classes that handle too many concerns.
* Example: An OrderService should handle order logic, not database access or email notifications. Those should be handled by other components like OrderRepository or EmailService.

public class User {  
 private String name;  
 private String email;  
 public User(String name, String email) {  
 this.name = name;  
 this.email = email;  
 }  
 public String getName() {  
 return name;  
 }  
 public String getEmail() {  
 return email;  
 }  
 // Bad practice: Handling file storage within the User class  
 public void saveToFile() {  
 try (FileWriter fileWriter = new FileWriter(name + ".txt")) {  
 fileWriter.write("Name: " + name + "\n");  
 fileWriter.write("Email: " + email + "\n");  
 System.out.println("User data saved to file successfully.");  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 public static void main(String[] args) {  
 User user = new User("John Doe", "john.doe@example.com");  
 user.saveToFile();  
 }

}

**3. Spring Boot Specific Guidelines**

* **Controller Layer:** Keep controllers **thin**. They should only handle HTTP-specific logic, like mapping requests to services and preparing the response. Business logic should be handled in services.



**Service Layer:** Services should be **stateless** and focus on **business logic**. Avoid direct interactions with databases or external services here. Let repositories and external clients handle persistence and external APIs.

**Repository Layer:** The repository layer should be focused on data access logic. Use **Spring Data JPA** repositories for CRUD operations and custom queries.

**Repository Layer:** The repository layer should be focused on data access logic. Use **Spring Data JPA** repositories for CRUD operations and custom queries.



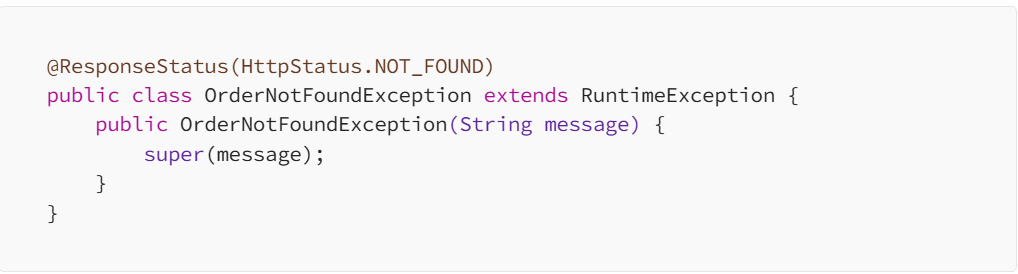
**Repository Layer:** The repository layer should be focused on data access logic. Use **Spring Data JPA** repositories for CRUD operations and custom queries.

**DTOs (Data Transfer Objects):** Use **DTOs** for **request/response payloads**. Keep them **separate from domain entities** to avoid exposing internal structure.



**4. Error Handling**

* **Use Custom Exceptions:** Create **custom exceptions** to handle application-specific errors. Avoid generic exceptions like Exception or RuntimeException.



**Use @ExceptionHandler for Global Error Handling:** Define a global exception handler using @ControllerAdvice to handle all exceptions and return meaningful responses.

@ControllerAdvice  
public class GlobalExceptionHandler {  
 @ExceptionHandler(OrderNotFoundException.class)  
 public ResponseEntity<ErrorResponse> handleOrderNotFound(OrderNotFoundException ex) {  
 ErrorResponse error = new ErrorResponse("Order Not Found", ex.getMessage());  
 return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(error);  
 }  
}

**5. Testing**

* **Write Unit Tests:** Always write unit tests for business logic in services using **JUnit** or **TestNG**. Use **Mockito** for mocking dependencies.

@ExtendWith(MockitoExtension.class)  
public class OrderServiceTest {  
 @Mock  
 private OrderRepository orderRepository;  
 @InjectMocks  
 private OrderService orderService;  
 @Test  
 void testCreateOrder() {  
 OrderRequest request = new OrderRequest(1L, Arrays.asList(101L, 102L));  
 when(orderRepository.save(any(Order.class))).thenReturn(new Order(1L));  
 Order result = orderService.createOrder(request);  
 assertNotNull(result);  
 assertEquals(1L, result.getId());  
 }  
}

**Write Integration Tests for Spring Boot:** Use **@SpringBootTest** for integration tests to verify that the entire Spring context is wired up correctly.

@SpringBootTest  
@AutoConfigureMockMvc  
public class OrderControllerTest {  
 @Autowired  
 private MockMvc mockMvc;  
 @Test  
 public void testCreateOrder() throws Exception {  
 OrderRequest orderRequest = new OrderRequest(1L, Arrays.asList(101L, 102L));  
 mockMvc.perform(post("/orders")  
 .contentType(MediaType.APPLICATION\_JSON)  
 .content(new ObjectMapper().writeValueAsString(orderRequest)))  
 .andExpect(status().isCreated())  
 .andExpect(jsonPath("$.orderId").exists());   
 } }

**6. Code Formatting & Readability**

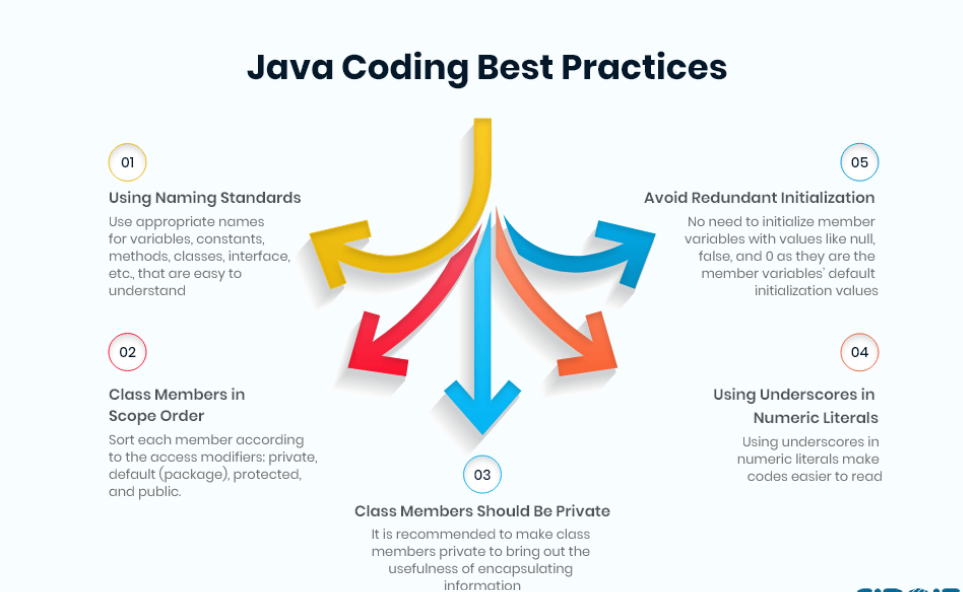
* **Consistent Indentation:** Use 4 spaces for indentation (do not use tabs). Ensure that the code is consistently indented, even in nested blocks.
* **Limit Line Length:** Aim for **80–100 characters per line**. Long lines should be broken into smaller ones to enhance readability.
* **Commenting:** Only comment **why** something is done, not **what** is being done. The code itself should describe **what** is happening through descriptive names and structure. Use comments sparingly.

// Calculate the total price including discounts  
BigDecimal totalPrice = calculateTotal(order);

* **Avoid Large Classes/Files:** Keep classes and files small, ideally under **200 lines**. If a class or file grows too large, split it logically into smaller components.

**7. Avoiding Common Pitfalls**

* **Avoid Magic Numbers/Strings:** Replace hardcoded values with **constants** or **configuration values**.
* Example: Use MAX\_RETRIES instead of 5 or "timeout" instead of "5000".
* **Minimize the Use of Static Methods:** Static methods are difficult to test and mock. Favor instance methods and dependency injection (through **Spring**).
* **Avoid Nested Loops/Conditionals:** If your logic has deeply nested loops or conditionals, consider refactoring it into smaller, more focused methods.

****