# Guidelines

1. **Naming conventions**: Follow standard Java naming conventions.
   1. Class names: PascalCase (e.g., UserRepository)
   2. Method names: camelCase (e.g., findAllUsers)
   3. Variable names: camelCase (e.g., firstName)
   4. Constants: UPPER\_SNAKE\_CASE (e.g., MAX\_USERS)
2. **Code organization**: Organize code using packages to group related functionalities.
3. com.example.application
4. ├── api
5. ├── config
6. ├── controller
7. ├── util
8. └── domain (e.g. "users" or "vulnerabilities" or "notifications")
9. ├── converter
10. ├── dao
11. ├── model
12. └── persistence (or repository)
13. ├── mapper
14. ├── model

└── service

1. **Indentation and spacing**: Use consistent indentation and spacing throughout the code.
   1. Indent with 4 spaces
   2. Use a single blank line to separate methods, and to logically group sections within a method
   3. Include a space before and after operators, such as =, +, and ==
2. **Comments and documentation**: Use comments to explain complex or non-obvious code. Write JavaDoc comments for public APIs.
3. /\*\*
4. \* Retrieves all users in the system.
5. \*
6. \* @return a list of all users
7. \*/
8. public List<User> findAllUsers() {
9. ...

}

1. **Use Spring annotations**: Leverage Spring Boot annotations for dependency injection and configuration.
   1. @Controller, @RestController: for defining controller classes
   2. @Service: for defining service classes
   3. @Repository: for defining repository classes
   4. @Autowired: for injecting dependencies
2. **Error handling**: Use exception handling to handle errors and provide meaningful error messages. Avoid using standard Exceptions and create custom Exceptions, clearly stating the Exception case. ( E.g. UserNotFoundException ).
3. try {
4. // code that may throw an exception
5. } catch (SomeException e) {
6. // handle the exception, e.g., log an error message or return a custom error response

}

And with the code below you can create custom exceptions:

public class UserNotFoundException extends RuntimeException {

public UserNotFoundException(String message) {

super(message);

}

public UserNotFoundException(String message, Throwable cause) {

super(message, cause);

}

}

1. **Logging**: Use logging frameworks like SLF4J with Logback for logging purposes.
2. import org.slf4j.Logger;
3. import org.slf4j.LoggerFactory;
4. public class UserController {
5. private static final Logger logger = LoggerFactory.getLogger(UserController.class);
6. public User getUser(Long userId) {
7. logger.info("Fetching user with id: {}", userId);
8. ...
9. }

}

1. **Unit testing**: Write unit tests for your code using frameworks like JUnit, Mockito, and Jupiter.
2. import static org.junit.jupiter.api.Assertions.assertEquals;
3. @Test
4. public void testFindAllUsers() {
5. List<User> users = userService.findAllUsers();
6. assertEquals(5, users.size());

}

1. **Consistent code style**: Use a consistent code style throughout your codebase. You can enforce this using tools like Checkstyle, PMD, or FindBugs.
2. **Use Lombok**: Reduce boilerplate code in your Java classes using Lombok's annotations like @Data, @AllArgsConstructor, and @NoArgsConstructor.
3. **Minimize visibility**: Declare variables, methods, and classes with the minimum necessary visibility (e.g., private, protected, public). Prefer using private and provide accessor methods (getters and setters) when necessary.
4. **Use interfaces**: Favor interfaces over abstract classes for more flexible and modular code. Interfaces allow you to define contracts without enforcing a specific class hierarchy.
5. **Prefer composition over inheritance**: Use composition to reuse code and create more flexible and maintainable systems. Inheritance should be reserved for cases where there is a clear "is-a" relationship.
6. **Immutable objects**: Whenever possible, create immutable objects that cannot be modified after being instantiated. Immutable objects are simpler to reason about, easier to share across threads, and more secure.
7. **Use final keyword**: Use the final keyword to create constants, prevent method overriding, and declare classes that should not be subclassed.
8. **Avoid using null**: Minimize the use of null values in your code to reduce the chance of NullPointerException. Use Java 8's Optional class to represent the absence of a value when appropriate.
9. **Follow the single responsibility principle**: Each class and method should have a single, well-defined responsibility. This makes the code easier to understand, test, and maintain.
10. **Use meaningful names**: Choose meaningful and descriptive names for your classes, methods, and variables. Avoid abbreviations and single-letter names, except for loop indices or lambda parameters.
11. **Initialize collections with proper capacity**: When you know the final size of a collection, initialize it with the correct capacity to avoid unnecessary resizing and improve performance.
12. **Use Java standard libraries**: Leverage Java's standard libraries for common operations like working with collections, handling dates and times, or performing I/O operations.
13. **Use the enhanced for loop**: When iterating over a collection or an array, use the enhanced for loop for better readability.
14. **Follow the "Fail Fast" principle**: Validate method arguments and object state as early as possible to detect programming errors and throw appropriate exceptions.
15. **Use factory methods**: When appropriate, use factory methods to create objects instead of constructors. Factory methods can have descriptive names and return subtypes or interface implementations.
16. **Avoid premature optimization**: Focus on writing clear and maintainable code first. Optimize your code only after identifying performance issues through profiling or other analysis.
17. **Use resource management**: When working with resources like files, network connections, or database connections, use the try-with-resources statement to ensure proper resource management and avoid resource leaks.
18. **Don't repeat yourself:** E.g. Only qualify field access with this if it is necessary (i.e. local variable with the same name). Otherwise do not use the this keyword.
19. **Don't use deprecated resources**
20. **One true brace style (1TBS)**: For control structures like if, else, for, while, and do, use the One True Brace Style (1TBS) or "K&R style." This means placing the opening brace at the end of the line and the closing brace on a new line, aligned with the control structure.
21. **Limit nesting levels**: Deeply nested code can be difficult to read, understand, and maintain. To improve readability and maintainability, try to limit the number of nested control structures such as if, else, for, while, and switch statements. A commonly recommended maximum nesting depth is 3 to 4 levels. To reduce nesting levels, you can:
    1. Use guard clauses: Instead of nesting code inside a if statement, use guard clauses to handle special cases or invalid input and return early from the method.
    2. // Preferred
    3. public void processUser(User user) {
    4. if (user == null) {
    5. // Handle the null case and return early
    6. return;
    7. }
    8. // Continue processing the user
    9. }
    10. // Not recommended
    11. public void processUser(User user) {
    12. if (user != null) {
    13. // Process the user inside the nested block
    14. } else {
    15. // Handle the null case
    16. }

}

* 1. **Extract methods**: If you have a deeply nested block of code, consider extracting it into a separate method. This not only reduces nesting but also helps to break down complex methods into smaller, more focused, and reusable pieces.
  2. // Preferred
  3. public void processUsers(List<User> users) {
  4. for (User user : users) {
  5. processUser(user);
  6. }
  7. }
  8. private void processUser(User user) {
  9. // Process the user
  10. }
  11. // Not recommended
  12. public void processUsers(List<User> users) {
  13. for (User user : users) {
  14. // Process the user with nested code
  15. }
  16. }
  17. **Use functional programming**: With Java 8 and later, you can use functional programming techniques like streams and lambda expressions to simplify complex nested structures.
  18. // Preferred
  19. users.stream()
  20. .filter(user -> user.getAge() >= 18)
  21. .forEach(user -> processUser(user));
  22. // Not recommended
  23. for (User user : users) {
  24. if (user.getAge() >= 18) {
  25. processUser(user);
  26. }}