**Best Practices**

[General Principles 4](#_Toc207781194)

[Readability First 4](#_Toc207781195)

[Consistent Naming 4](#_Toc207781196)

[Common Formatting Style 5](#_Toc207781197)

[Follow KISS, DRY, YAGNI principles 5](#_Toc207781198)

[Class level design 6](#_Toc207781199)

[Single Responsibility Principle 6](#_Toc207781200)

[Open-Closed Principle 7](#_Toc207781201)

[Liskov Substitution Principle 8](#_Toc207781202)

[Interface Segregation Principle 10](#_Toc207781203)

[Dependency Inversion Principle 11](#_Toc207781204)

[Composition over Inheritance 12](#_Toc207781205)

[Exception Handling 14](#_Toc207781206)

[Consistency in Exception Handling 14](#_Toc207781207)

[Avoid exceptions to control flow of program 15](#_Toc207781208)

[Catch specific exceptions instead of generic 15](#_Toc207781209)

[Do not suppress exceptions 16](#_Toc207781210)

[Use try-with-resources 16](#_Toc207781211)

[Define custom exception hierarchies for your domain 17](#_Toc207781212)

[Tests 17](#_Toc207781213)

[AAA Pattern 17](#_Toc207781214)

[Single assertion per test 18](#_Toc207781215)

[Use @ParametrizedTest to avoid duplication 18](#_Toc207781216)

[Use annotations for cleaner code @Mock, @InjectMocks, @Captor 18](#_Toc207781217)

[Stream API 19](#_Toc207781218)

[Avoid overusing parallel in the coding process 19](#_Toc207781219)

[Use filter before map 20](#_Toc207781220)

[Prefer method references instead of lambdas 20](#_Toc207781221)

[Use streams of primitives where necessary 21](#_Toc207781222)

[Side effects break the functional paradigm 21](#_Toc207781223)

[Optional API 21](#_Toc207781224)

[Never assign null to an optional variable 21](#_Toc207781225)

[Don't overuse Optional by chaining its methods just to get a value 22](#_Toc207781226)

[Avoid boxing and unboxing: use non-generic Optional 22](#_Toc207781227)

[Reject wrapped values using filter() 22](#_Toc207781228)

[Use map() or flatMap() to transform value - no need to use isPresent() 23](#_Toc207781229)

[Use stream() to treat the Optional instance as Stream 23](#_Toc207781230)

[Avoid using an isPresent() and get() pair 23](#_Toc207781231)

[Don’t use orElse() for returning a computed value 24](#_Toc207781232)

[Don’t use Optional in Entity Fields 24](#_Toc207781233)

[Use Optional only for return types, not for fields or parameters 24](#_Toc207781234)

[Lambdas 25](#_Toc207781235)

[Prefer standard functional interfaces over custom one 25](#_Toc207781236)

[Use method references 25](#_Toc207781237)

[Avoid blocks of code in lambda’s body 26](#_Toc207781238)

[Avoid specifying parameter type 26](#_Toc207781239)

[Collections API 26](#_Toc207781240)

[Always try to use interface instead of a class while declaring any collection 26](#_Toc207781241)

[Keep immutable objects as a keys in Map collection 27](#_Toc207781242)

[Avoid modifying collections while iterating over them, for this purpose you can use Iterator 27](#_Toc207781243)

[Specify collection size if possible 28](#_Toc207781244)

[Use concurrent collections only when needed 28](#_Toc207781245)

[For collections that are not intended to be modified after creation, use immutable collections 28](#_Toc207781246)

[Generics 29](#_Toc207781247)

[Don’t use raw types 29](#_Toc207781248)

[Use bounded type parameters 29](#_Toc207781249)

[Prefer interfaces to implementations in generic types 30](#_Toc207781250)

[Use PECS rule – Producer Extends, Consumer Super 30](#_Toc207781251)

[Spring 31](#_Toc207781252)

[Proper package style 31](#_Toc207781253)

[Use constructor injection with Lombok 31](#_Toc207781254)

[Handle exceptions using @ControllerAdvice 32](#_Toc207781255)

[Keep controllers thin 33](#_Toc207781256)

[Use appropriate @ResponseStatus 33](#_Toc207781257)

[Hibernate 34](#_Toc207781258)

[Filtering Data in Java Instead of the Database 34](#_Toc207781259)

[Fetching the Entire Entity When You Only Need a Reference 34](#_Toc207781260)

[Eagerly Loading Collections You Don’t Need 35](#_Toc207781261)

[Updating Entities Without Explicit Transactions 35](#_Toc207781262)

[Blindly Using CascadeType.ALL 36](#_Toc207781263)

[Manually Updating Both Sides of a Bidirectional Relationship 37](#_Toc207781264)

[Leaving EntityManager Open in the Web Layer 38](#_Toc207781265)

[Fetching Large Data Sets Without Pagination 38](#_Toc207781266)

[Exposing Entity Objects Directly in API Responses 38](#_Toc207781267)

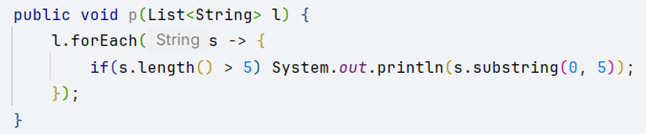
# General Principles

## Readability First

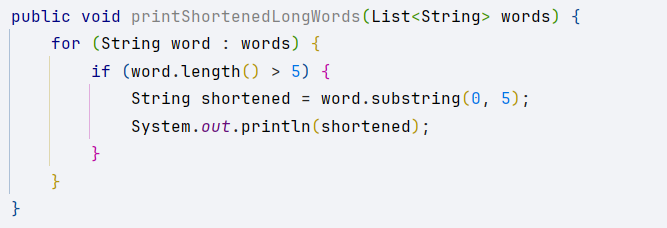
Code is read much more often than it is written. Write code so that it is easily understandable by others (and your future self). Prioritize clarity over clever tricks or overly concise solutions.

*Example:*

Bad:



Good:

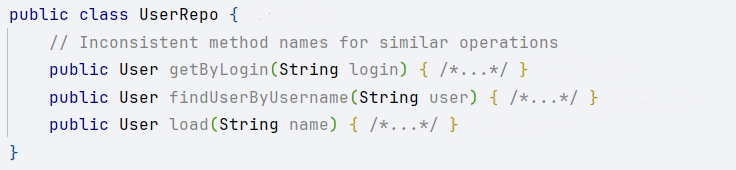


## Consistent Naming

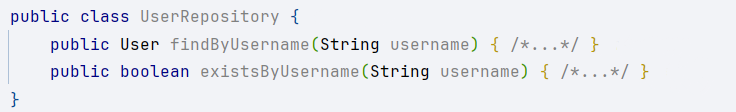
Follow consistent naming conventions throughout your project. Name variables, classes, and methods according to what they represent, and stick to common standards.

*Example:*

Bad:



Good:

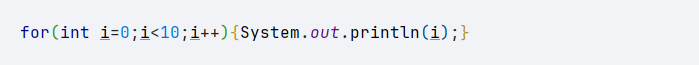


## Common Formatting Style

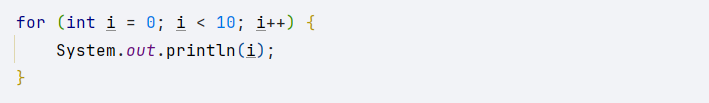
Adopt a common code formatting style for indentation, spacing, braces, etc., and use it consistently throughout the codebase.

*Example:*

Bad:



Good:

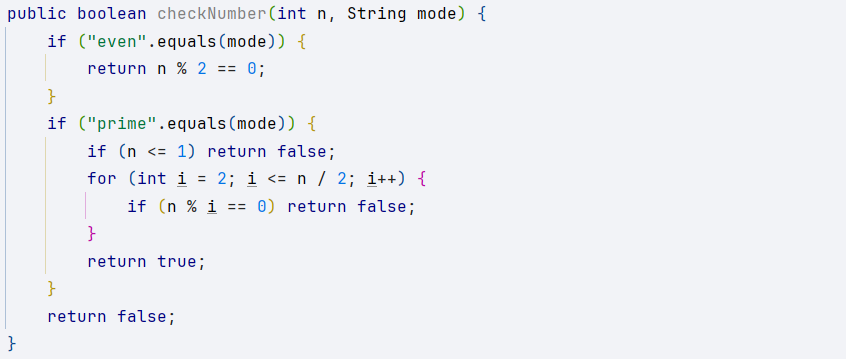


## Follow KISS, DRY, YAGNI principles

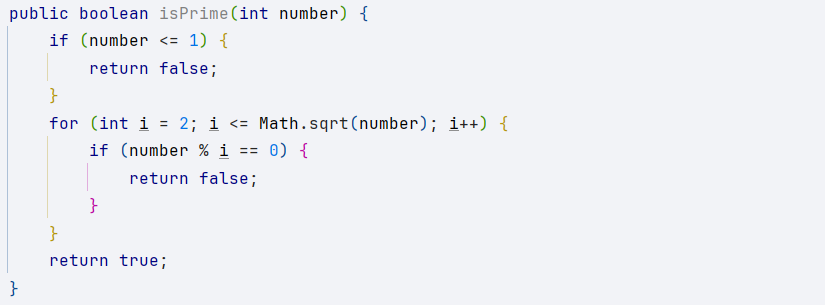
* KISS (Keep It Simple, Stupid): Write simple, straightforward solutions; avoid unnecessary complexity.
* DRY (Don’t Repeat Yourself): Eliminate code duplication by abstracting common logic.
* YAGNI (You Aren’t Gonna Need It): Don’t implement functionality until it’s actually needed.

*Example:*

Bad:



Good:



# Class level design

## Single Responsibility Principle

Each class/module should have exactly one reason to change — it owns one cohesive responsibility. SRP improves readability, testability, reusability, and reduces merge conflicts by isolating concerns.

*Violations*

* “God Service/Controller”: thousands of lines, many @Autowired fields, mixed concerns (validation + business + persistence + messaging).
* Frequent, unrelated changes to the same class by different teams (auth team, email team, reporting team all touch it).
* Conditionals or feature flags scattering responsibilities: if (emailEnabled) …; if (auditEnabled) …;
* Hard-to-test classes require multiple external systems to be available for a single unit test.

*Example:*

Bad:



Good:





## Open-Closed Principle

* Software entities should be open for extension but closed for modification.
* We should be able to add new behavior without changing existing tested code.
* Encourages polymorphism, inheritance, strategy, or composition over if/else chains.

*Violations:*

* Large if/else or switch statements deciding behavior based on a type/enum.
* Adding a new case requires editing existing classes instead of adding new ones.
* Business rules scattered across multiple conditionals.
* Tests break often when adding a new “variant.”

*Example:*

Bad:



Good:



## Liskov Substitution Principle

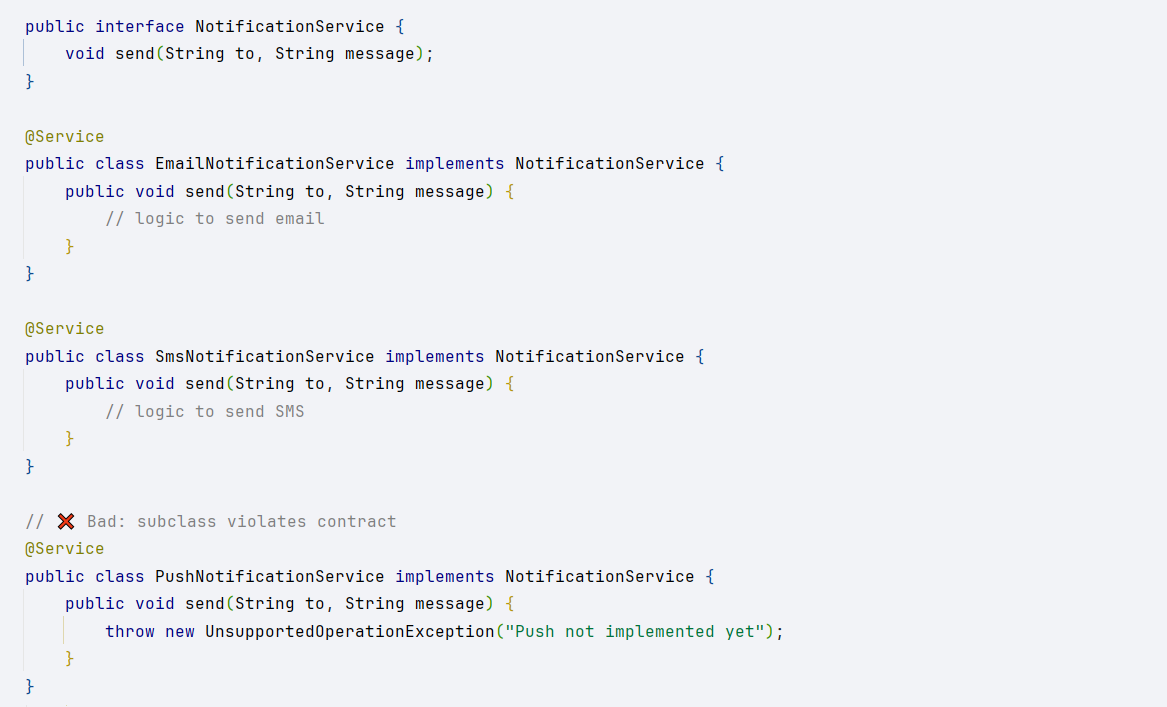
* Subtypes must be substitutable for their base types without altering correctness.
* Clients should not need to know whether they’re working with the parent class or a subclass.
* “If S is a subtype of T, then objects of type T may be replaced with objects of type S without breaking the program.”

*Violations:*

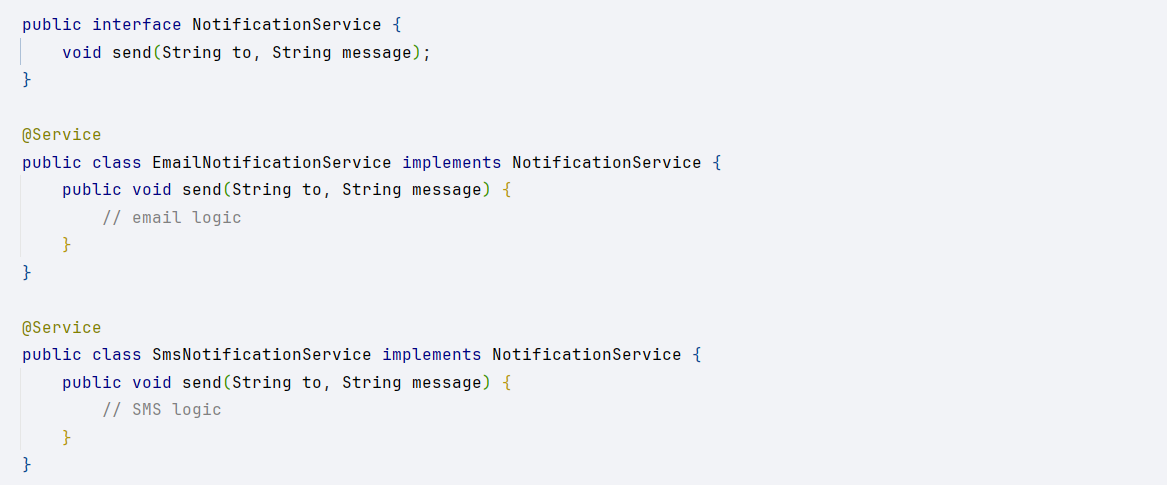
* Subclass throws UnsupportedOperationException for base class methods.
* Overridden method changes expected behavior (postconditions weakened, invariants broken).
* Client code filled with instanceof checks to handle subtypes differently.

*Example:*

Bad:



Good:



## Interface Segregation Principle

* Clients should not be forced to depend on methods they do not use.
* Instead of one “fat” interface, provide multiple small, focused interfaces.
* Leads to higher cohesion, easier testing, and better substitution.

*Violations:*

* Interfaces with too many methods (a “God interface”).
* Implementations throwing UnsupportedOperationException for unused methods.
* Classes implementing unrelated behaviors just to satisfy an interface.
* Test classes need to mock/stub methods they don’t care about.

*Example:*

Bad:



Good:



## Dependency Inversion Principle

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Details (implementations) should depend on policies (interfaces), not the other way around.
* Promotes loose coupling and easier substitution/testing.

*Violations:*

* Services directly instantiate dependencies with new.
* High-level classes depend on framework details instead of domain abstractions.
* Swapping an implementation requires modifying business code.

*Example:*

Bad:



Good:



## Composition over Inheritance

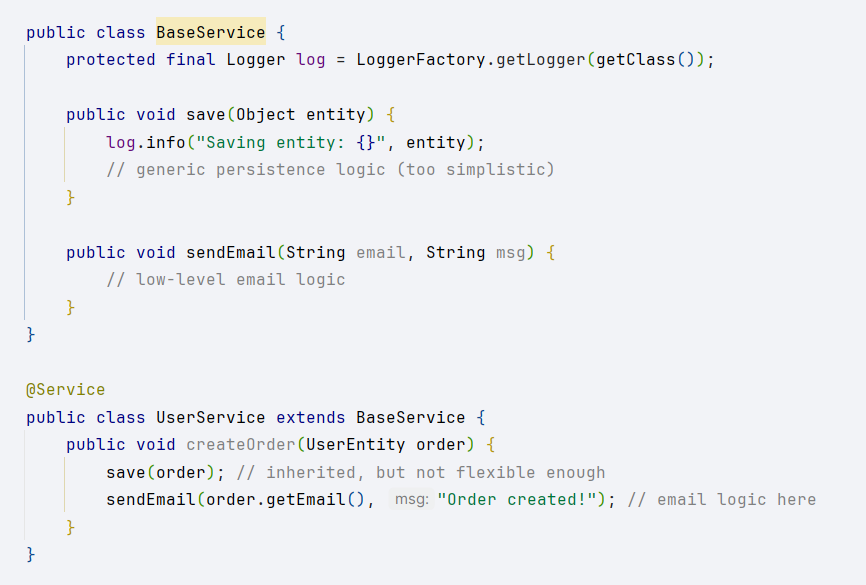
* Favor composition (HAS-A relationship) over inheritance (IS-A) when designing classes.
* Composition delegates behavior to other objects instead of hardcoding it in a parent class.
* Benefits: flexibility, reuse, easier testing, and fewer fragile hierarchies.

*Violations:*

* Deep or wide class hierarchies (AbstractBaseService → BaseService → CustomService).
* Overridden methods that change parent behavior in unexpected ways.
* Child classes inheriting methods/fields they don’t need (“fat” base class).
* instanceof checks to differentiate subclass behaviors.
* Difficulty introducing new behavior without breaking parent or siblings.

*Example:*

Bad:



Good:



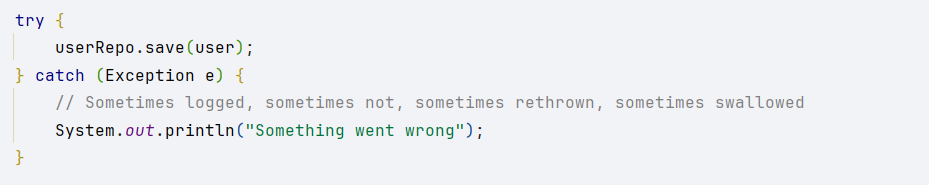
# Exception Handling

## Consistency in Exception Handling

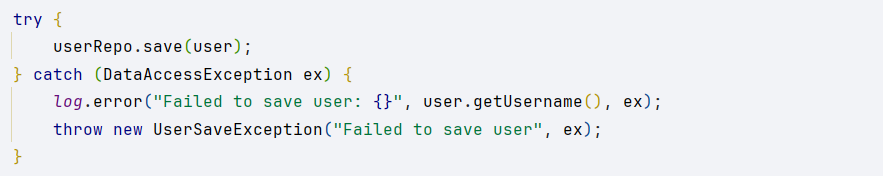
Adopt a standard approach to catching and handling exceptions across your codebase, so all errors are managed in a predictable way.

*Example:*

Bad:



Good:



## Avoid exceptions to control flow of program

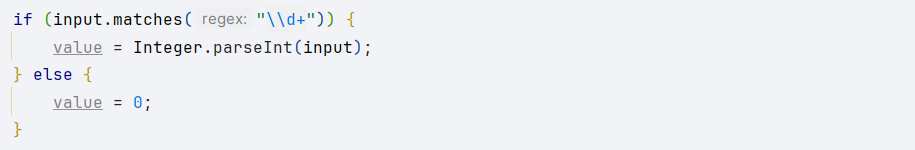
Don’t use exceptions for expected situations, such as checking if a file exists. Use regular control flow (if/else) for that.

*Example:*

Bad:



Good:

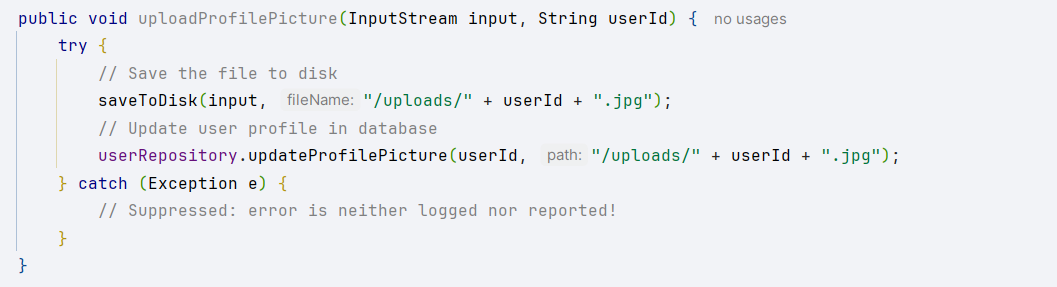


## Catch specific exceptions instead of generic

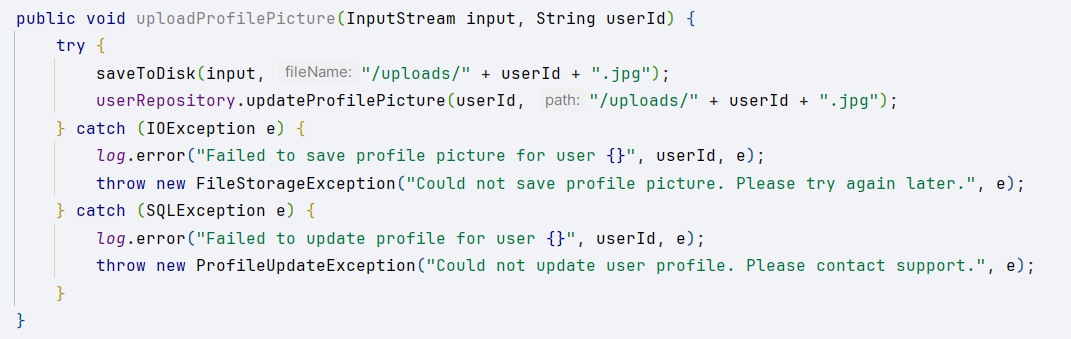
Catch only the exceptions you can actually handle. Catching generic Exception hides bugs and makes debugging harder.

*Example:*

Bad:



Good:

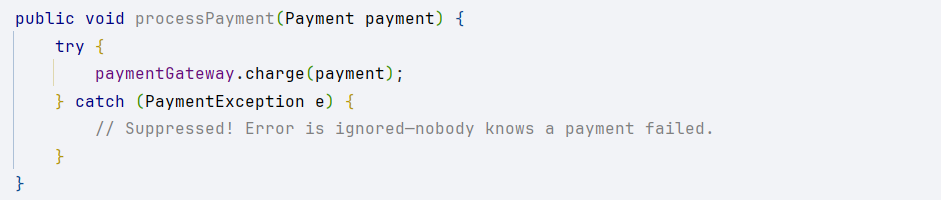


Do not suppress exceptions

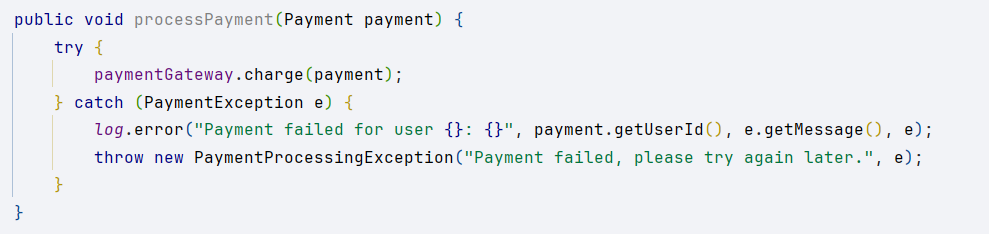
Never ignore exceptions. At minimum, log them; at best, handle or recover gracefully.

*Example:*

Bad:



Good:



## Use try-with-resources

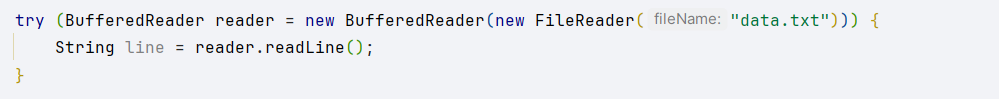
When working with resources (like streams, connections), always use try-with-resources to ensure they are closed automatically—even if an exception occurs.

*Example:*

Bad:



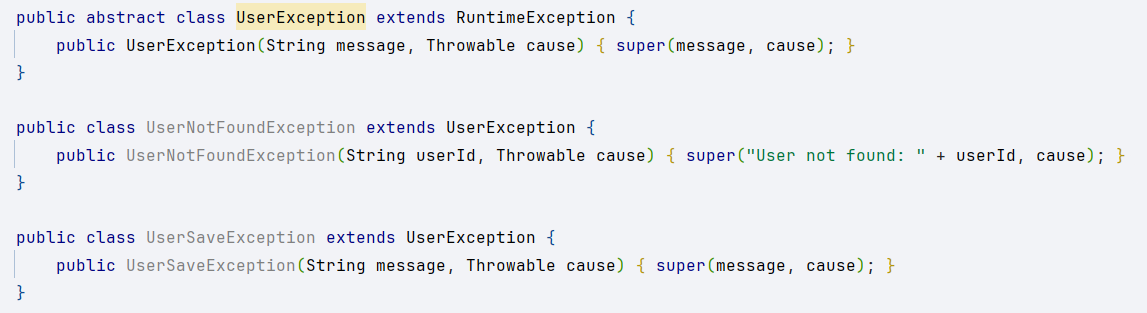
Good:



## Define custom exception hierarchies for your domain

Create your own exceptions (possibly organized in a hierarchy) to represent domain-specific error cases. This makes your error handling precise and meaningful.

*Example:*



<https://www.geeksforgeeks.org/java/best-practices-to-handle-exceptions-in-java/>

# Tests

## AAA Pattern

Structure your tests in three sections:

* Given (Arrange): Set up data and dependencies.
* When (Act): Call the method under test.
* Then (Assert): Verify the result.
* *Example:*



## Single assertion per test

A single test should verify one behavior. Multiple assertions for different behaviors make it harder to pinpoint what failed.

*Example:*



## Use @ParametrizedTest to avoid duplication

Use parameterized tests to run the same test logic with different input data, reducing code duplication and improving coverage.

*Example:*



## Use annotations for cleaner code @Mock, @InjectMocks, @Captor

Use Mockito annotations to simplify the creation of mocks and inject them into the system under test. This reduces boilerplate code and improves readability.

*Example:*



<https://www.linkedin.com/pulse/best-practices-using-mockito-junit-junior-nakamura-eamef/>

<https://www.linkedin.com/pulse/best-practices-unit-testing-java-mockito-junit-gabriel-sobreira-g69ef/>

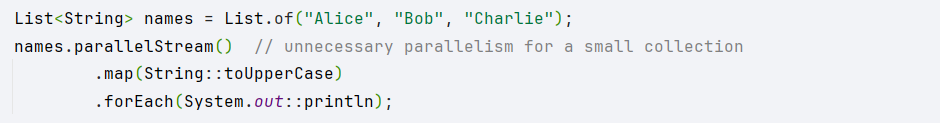
# Stream API

Avoid overusing parallel in the coding process

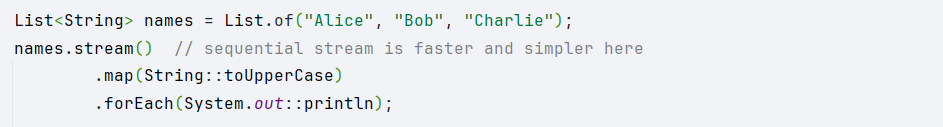
Don’t use *parallelStream()* by default. Parallel streams are only beneficial for CPU-intensive, non-blocking operations on large collections. Overuse can actually harm performance due to overhead and thread contention.

*Example:*

Bad:



Good:



## Use filter before map

Apply *filter()* before *map()* to process fewer elements. This improves performance and readability by ensuring you only transform relevant items.

*Example:*

Bad:



Good:



## Prefer method references instead of lambdas

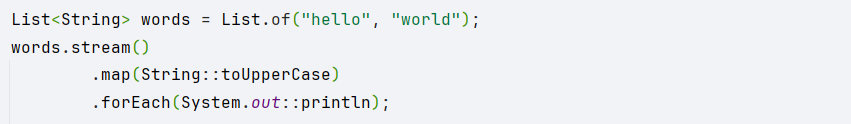
Where possible, use method references (String::toUpperCase) instead of lambdas (s -> s.toUpperCase). They are more concise and often easier to read.

*Example:*

Bad:



Good:

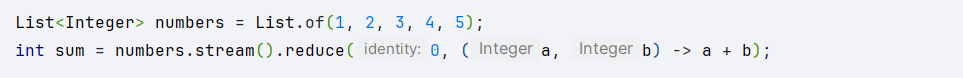


## Use streams of primitives where necessary

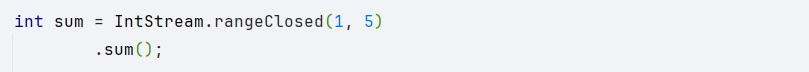
When processing large sequences of numbers or primitive data, use *IntStream*, *LongStream*, or *DoubleStream* to avoid unnecessary boxing and unboxing, which improves both speed and memory efficiency.

*Example:*

Bad:



Good:

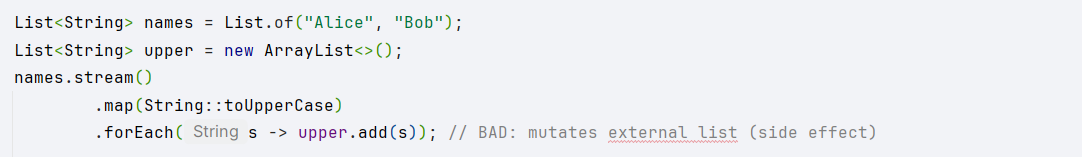


## Side effects break the functional paradigm

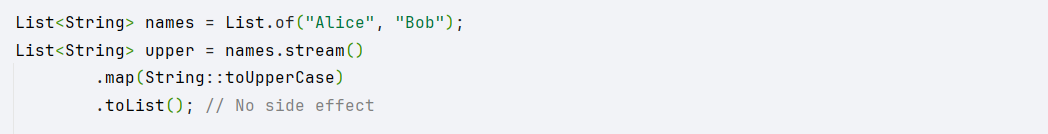
Don’t introduce side effects (like modifying external variables, or updating collections) inside stream operations. This can lead to bugs and unpredictable results.

*Example:*

Bad:



Good:



<https://readmedium.com/10-java-stream-tips-must-read-2063a84af3be>

# Optional API

## ****Never assign null to an optional variable****

Never assign or return *null* where an *Optional.empty()* should be used.

*Example:*

Bad:



Good:



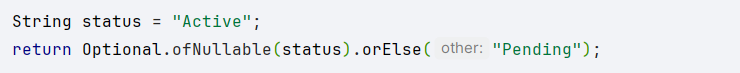
## Don't overuse Optional by chaining its methods just to get a value

Avoid excessive chaining just to retrieve a value — use Optional for meaningful, clear code, not as a replacement for simple if-else.

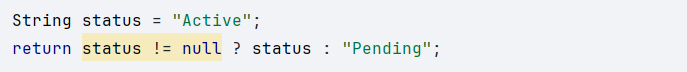
Developers can see a use case for Optional everywhere, by chaining its methods just for the single purpose of getting a value, and they forget about clarity, memory footprint, and being straightforward.

*Example:*

Bad:



Good:

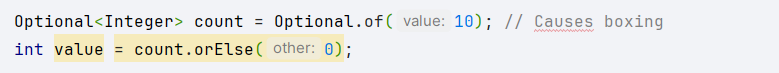


## Avoid boxing and unboxing: use non-generic Optional

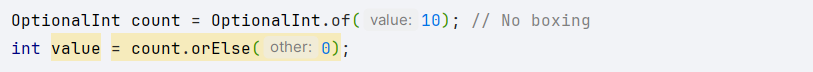
Use *OptionalInt*, *OptionalDouble*, etc., for primitives to avoid unnecessary boxing and unboxing.

*Example:*

Bad:



Good:

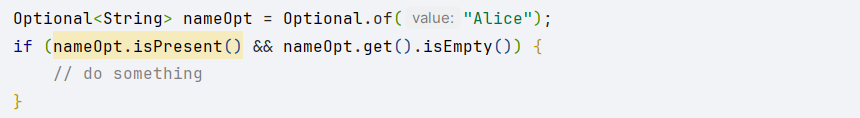


## ****Reject wrapped values using filter()****

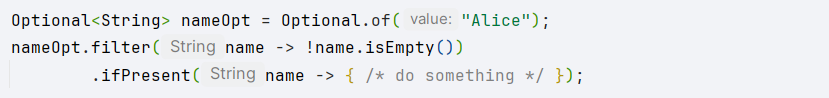
Use *filter()* to reject unwanted values, keeping code concise.

*Example:*

Bad:



Good:

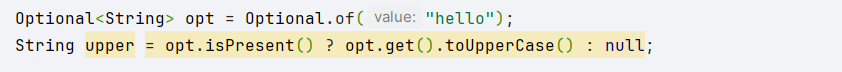


## ****Use**** map() ****or**** flatMap() ****to transform value - no need to use**** isPresent()

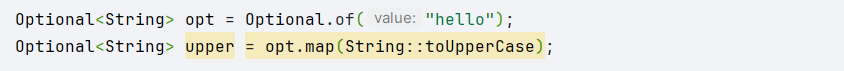
Transform the contained value using *map()* or *flatMap()*; avoid manual checks.

*Example:*

Bad:



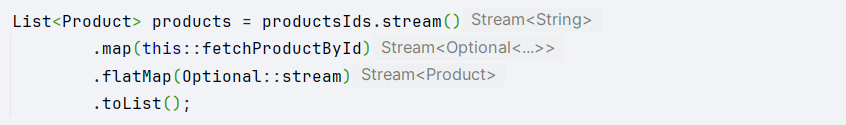
Good:



## ****Use**** stream() ****to treat the Optional instance as Stream****

From Java 9+, use *stream()* to elegantly process an Optional as a stream.

*Example:*



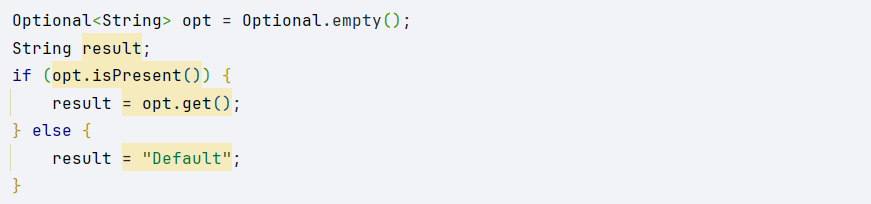
<https://dev.to/ivangavlik/how-to-use-the-optional-class-java-3pf5>

## **Avoid using an isPresent() and get() pair**

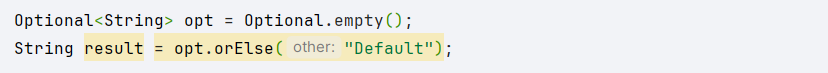
Don’t write if-get-else code when you can use *orElse()* or *orElseGet()*.

*Example:*

Bad:



Good:



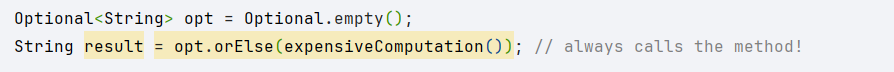
## ****Don’t use orElse() for returning a computed value****

A very important note to consider here is a probable performance penalty: The value returned by *orElse()* is always evaluated regardless of the optional value’s presence. So the rule here is to use *orElse()* when you have already preconstructed values and you don’t use an expensive computed value.

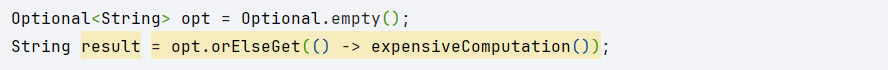
If the fallback is expensive to compute, use *orElseGet()* so the fallback is only evaluated if needed.

*Example:*

Bad:



Good:



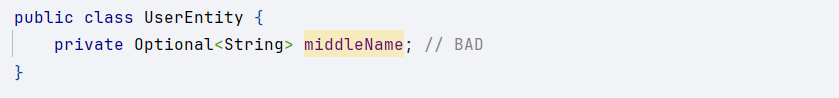
[**https://blogs.oracle.com/javamagazine/post/12-recipes-for-using-the-optional-class-as-its-meant-to-be-used**](https://blogs.oracle.com/javamagazine/post/12-recipes-for-using-the-optional-class-as-its-meant-to-be-used)

## Don’t use Optional in Entity Fields

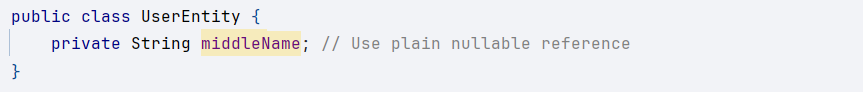
Never use Optional as a field in an entity, because it increases complexity for serialization frameworks like JPA, Jackson, etc. and leads to unnecessary wrapping and unwrapping during data handling.

*Example:*

Bad:



Good:



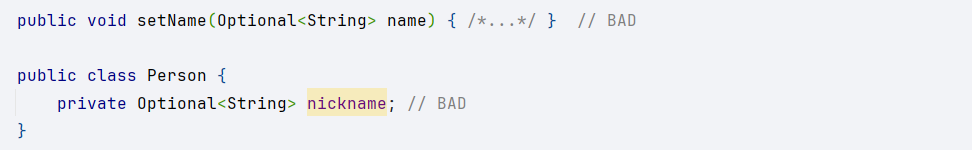
<https://medium.com/@arun.narayan/java-optional-anti-patterns-avoid-these-traps-for-cleaner-code-37cea5945265>

Use Optional only for return types, not for fields or parameters

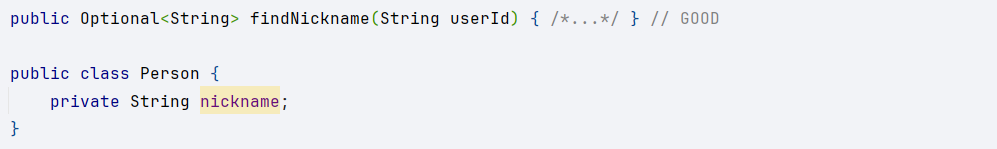
Optional should be used only as a return type to signal possible absence of a value, not as fields or method parameters.

*Example:*

Bad:



Good:



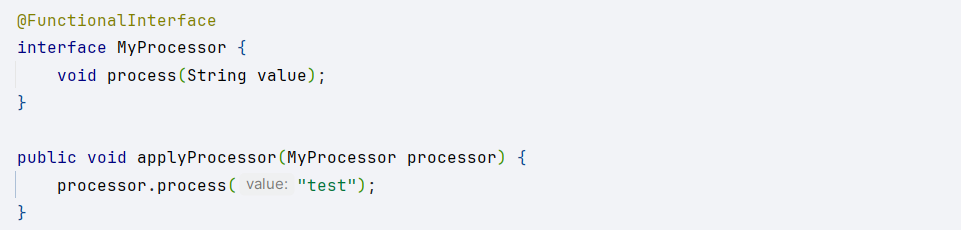
# Lambdas

## Prefer standard functional interfaces over custom one

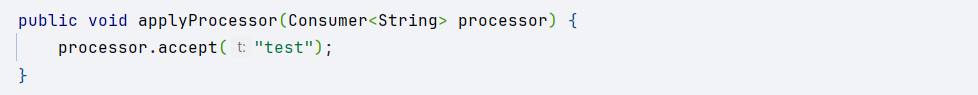
Use standard Java interfaces like *Function*, *Predicate*, *Consumer*, or *Supplier* instead of custom single-method interfaces.

*Example:*

Bad:



Good:

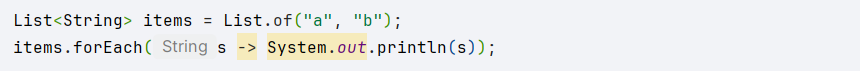


## Use method references

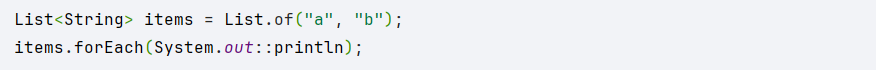
Use method references instead of equivalent lambdas for clarity and conciseness.

*Example:*

Bad:



Good:

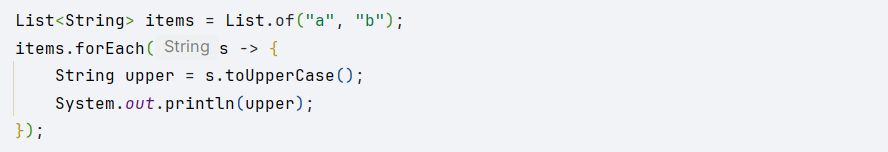


## Avoid blocks of code in lambda’s body

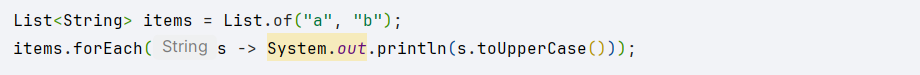
Keep lambda bodies simple—prefer single expressions over multiple statements or blocks.

*Example:*

Bad:



Good:

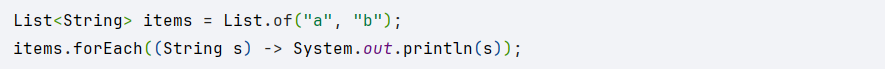


## Avoid specifying parameter type

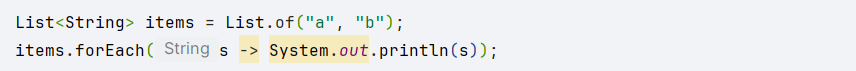
Let the compiler infer parameter types — don’t write them out unless absolutely necessary.

*Example:*

Bad:



Good:



<https://gelopfalcon.medium.com/best-practices-when-you-use-lambda-expressions-in-java-f51e96d44b25>

# Collections API

## Always try to use interface instead of a class while declaring any collection

Declare variables using the collection interface type (List, Set, Map, etc.) rather than concrete implementations. This increases flexibility and enables easy refactoring.

*Example:*

Bad:



Good:

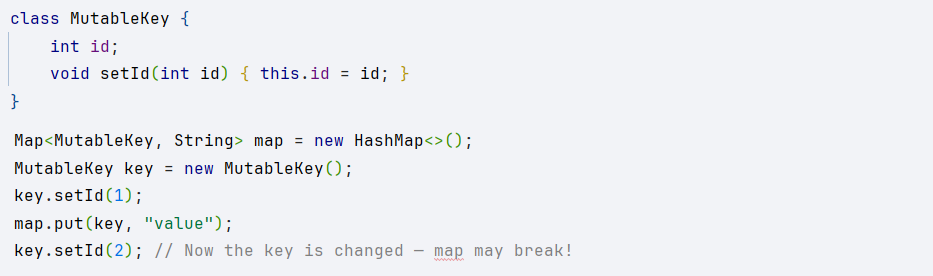


## Keep immutable objects as a keys in Map collection

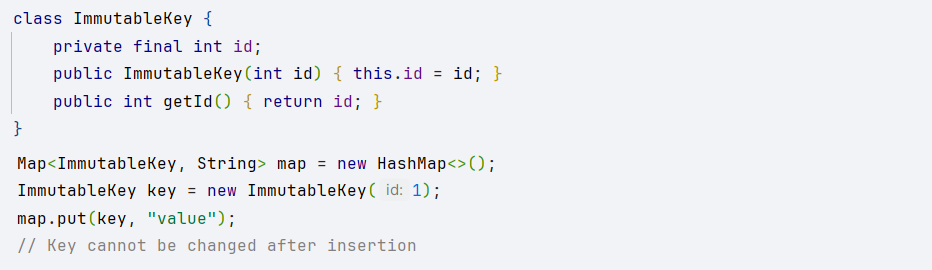
Use immutable objects as map keys to ensure correctness and predictable behavior, as mutable keys can cause issues in lookups and hashing.

*Example:*

Bad:



Good:

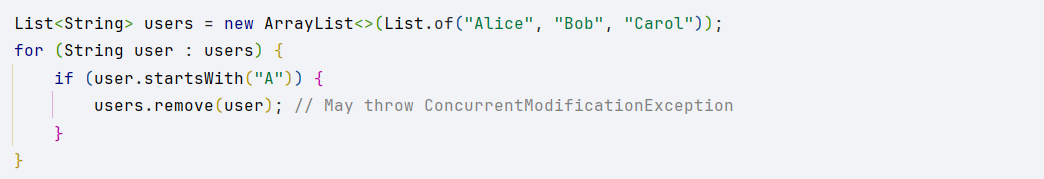


## Avoid modifying collections while iterating over them, for this purpose you can use Iterator

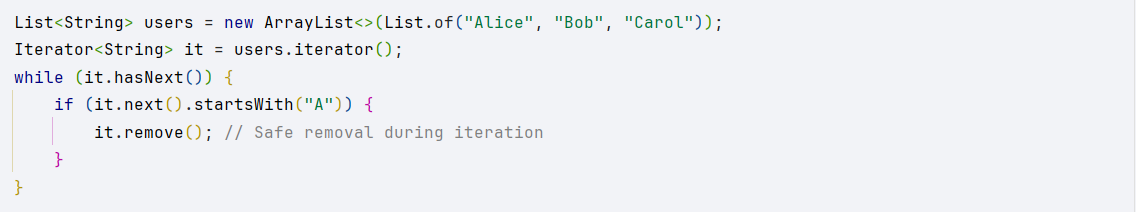
Don’t add or remove elements from a collection inside a loop unless you’re using an *Iterator*. This avoids *ConcurrentModificationException*.

*Example:*

Bad:



Good:

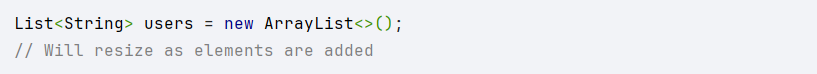


## Specify collection size if possible

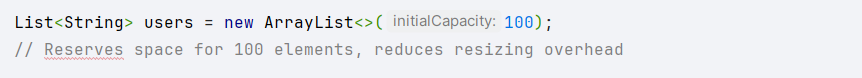
If you know the required size of a collection, specify it when creating to improve performance by avoiding unnecessary resizing.

*Example:*

Bad:



Good:



## Use concurrent collections only when needed

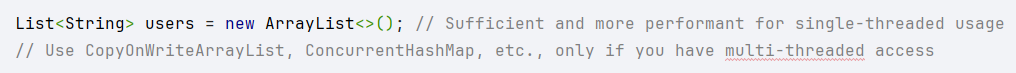
Use thread-safe (concurrent) collections only when actually required—regular collections are faster in single-threaded contexts.

*Example:*

Bad:



Good:

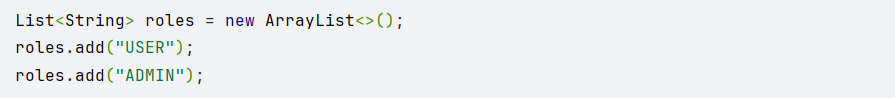


## For collections that are not intended to be modified after creation, use immutable collections

If a collection should not change after it’s created, use an unmodifiable or immutable variant to avoid accidental modification.

*Example:*

Bad:



Good:



<https://dev.to/soniajm/java-collections-best-practices-3j53>

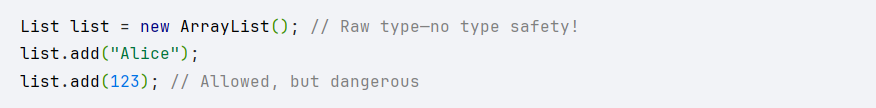
# Generics

## Don’t use raw types

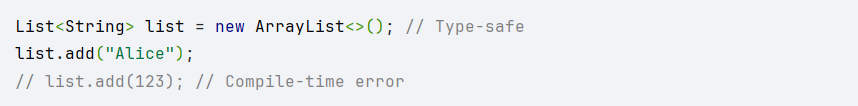
Always specify type parameters when using generics to ensure type safety and avoid runtime errors.

*Example:*

Bad:



Good:

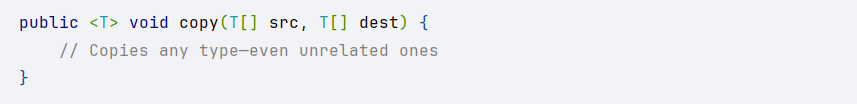


## Use bounded type parameters

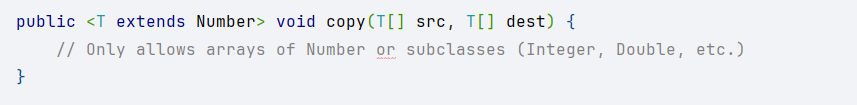
Use bounds (extends, super) to restrict type parameters when appropriate, ensuring that only valid types are used.

*Example:*

Bad:



Good:



## Prefer interfaces to implementations in generic types

Declare generics using interface types (like List<T>, Map<K, V>) instead of concrete implementations (like ArrayList<T>) to promote flexibility.

*Example:*

Bad:



Good:



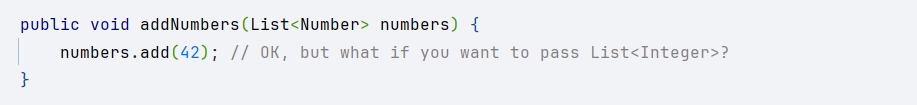
## Use PECS rule – Producer Extends, Consumer Super

Follow the PECS rule:

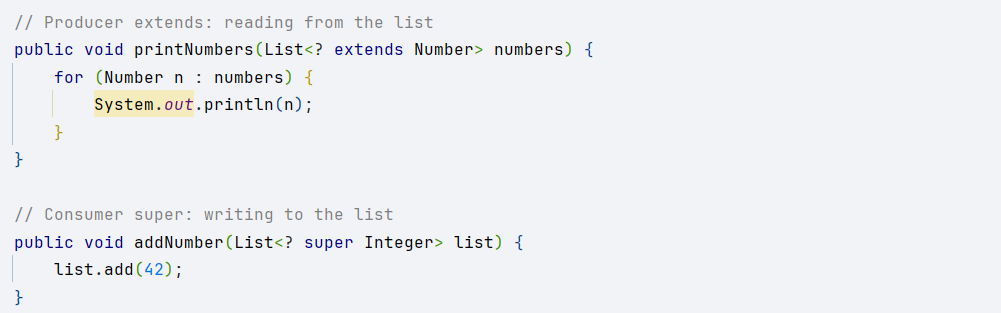
* Use *? extends T* when a structure produces objects of type T (read).
* Use *? super T* when a structure consumes objects of type T (write)

*Example:*

Bad:



Good:



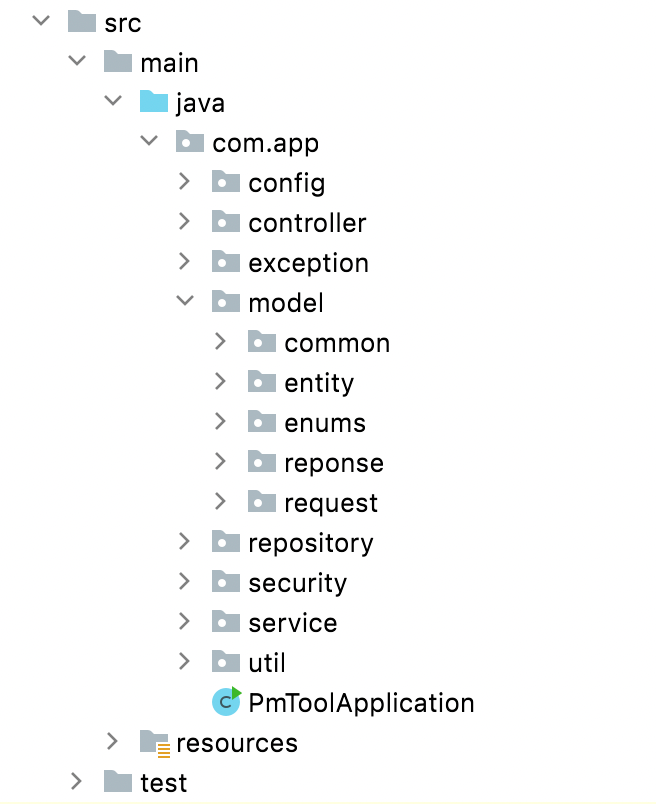
# Spring

## Proper package style

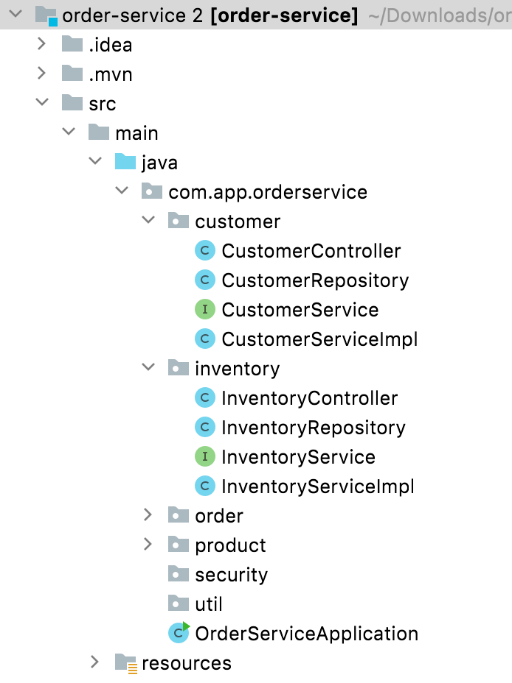
Organize your code in a standard, layered package structure (controller, service, repository, model, etc.) to separate concerns and improve maintainability.

*Example:*

Based on type:



Based on feature:

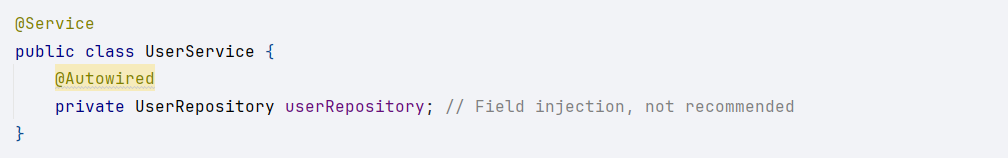


## Use constructor injection with Lombok

Prefer constructor-based dependency injection, and use Lombok’s @RequiredArgsConstructor to reduce boilerplate.

*Example:*

Bad:



Good:

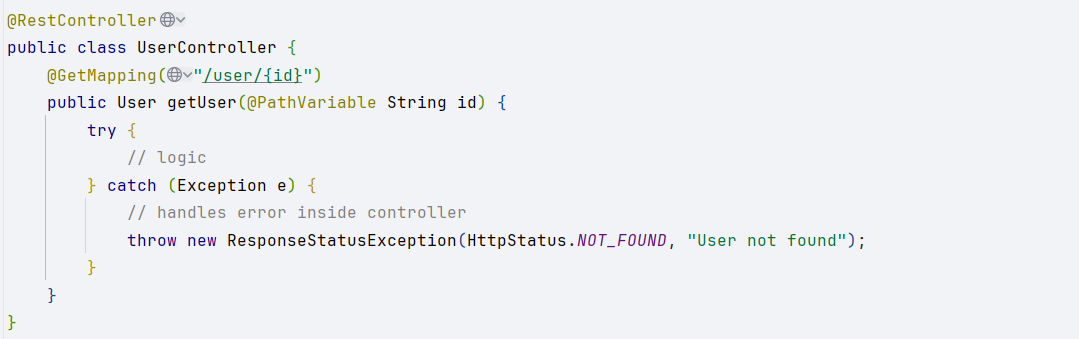


## Handle exceptions using @ControllerAdvice

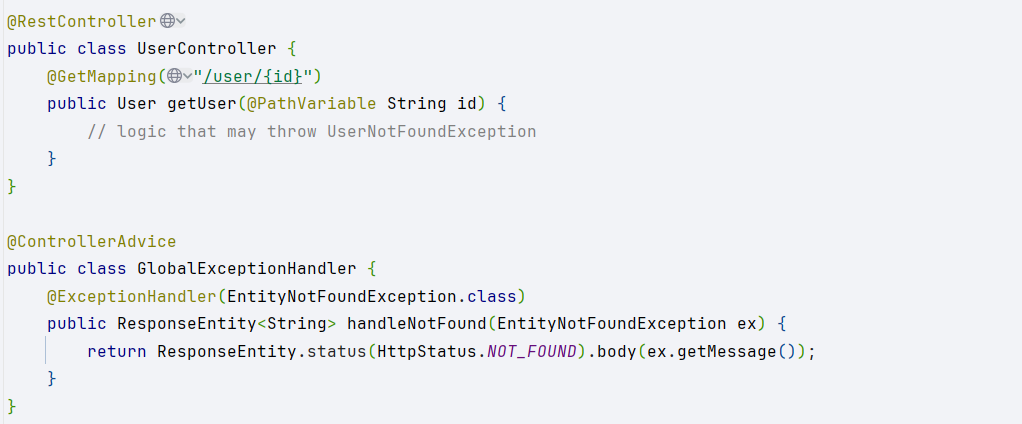
Centralize exception handling with @ControllerAdvice to keep controllers clean and error responses consistent.

*Example:*

Bad:



Good:



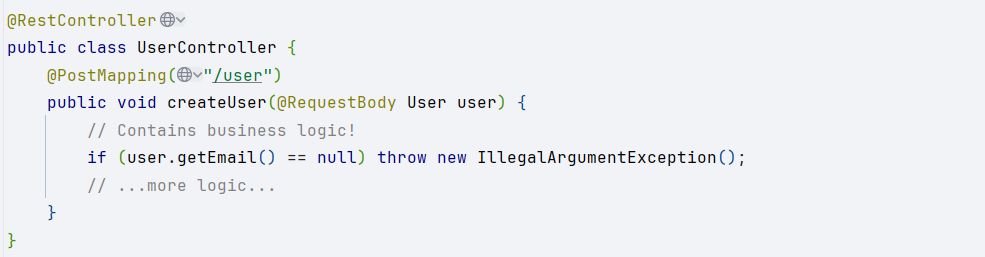
<https://medium.com/springboot-dev-hub/spring-boot-best-practices-for-developers-3f3bdffa0090>

## Keep controllers ****thin****

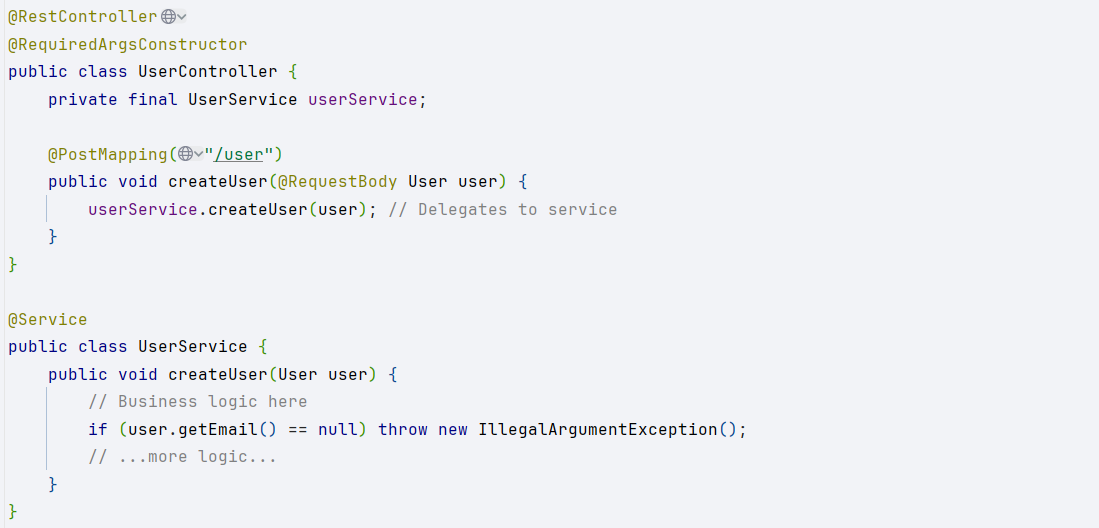
Controllers should only handle HTTP requests and delegate business logic to services. This separation keeps controllers simple and focused.

*Example:*

Bad:



Good:



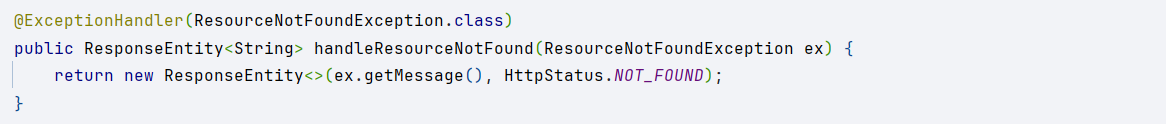
<https://medium.com/@sharmapraveen91/clean-code-coding-guidelines-java-spring-boot-api-36a88e588008>

## Use appropriate @ResponseStatus

Annotate custom exceptions with @ResponseStatus to return proper HTTP status codes for errors.

*Example:*

Bad:



Good:



<https://medium.com/@sharmapraveen91/handle-exceptions-in-spring-boot-a-guide-to-clean-code-principles-e8a9d56cafe8>

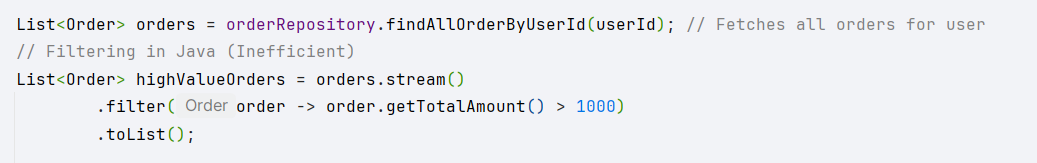
# Hibernate

## Filtering Data in Java Instead of the Database

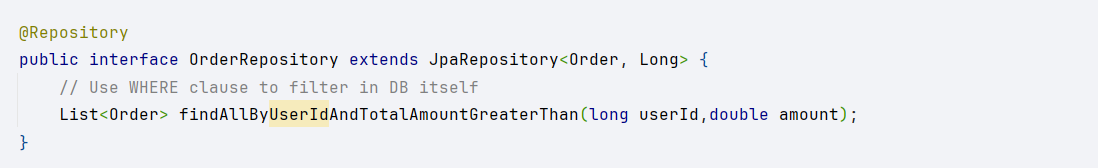
Always filter data at the database level when possible. Fetching too much data and filtering in Java is inefficient.

*Example:*

Bad:



Good:

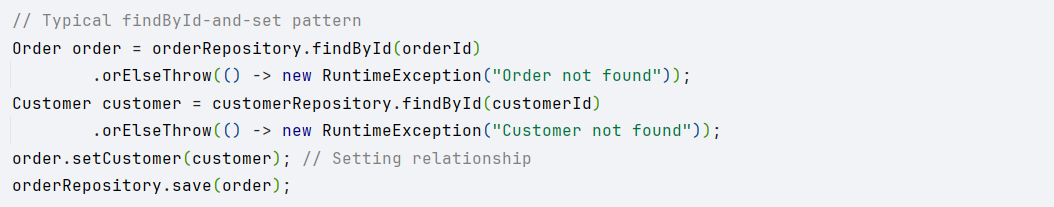


## Fetching the Entire Entity When You Only Need a Reference

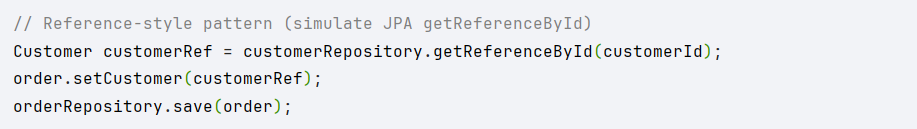
Use entity references or projections when you don’t need the full entity to avoid unnecessary data loading.

*Example:*

Bad:



Good:



## Eagerly Loading Collections You Don’t Need

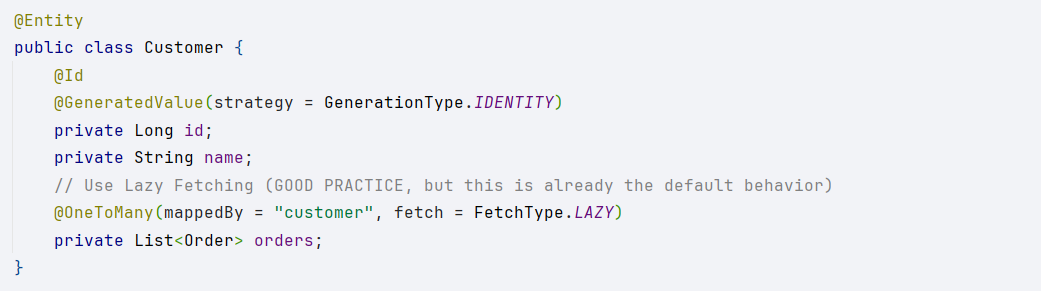
One of the most common performance pitfalls in Hibernate is eagerly loading collections when they are not needed. If an entity has a collection mapped with FetchType.EAGER, Hibernate will load all related entities automatically, even if they are not used, leading to unnecessary database queries and increased memory consumption.

*Example:*

Bad:



Good:



## Updating Entities Without Explicit Transactions

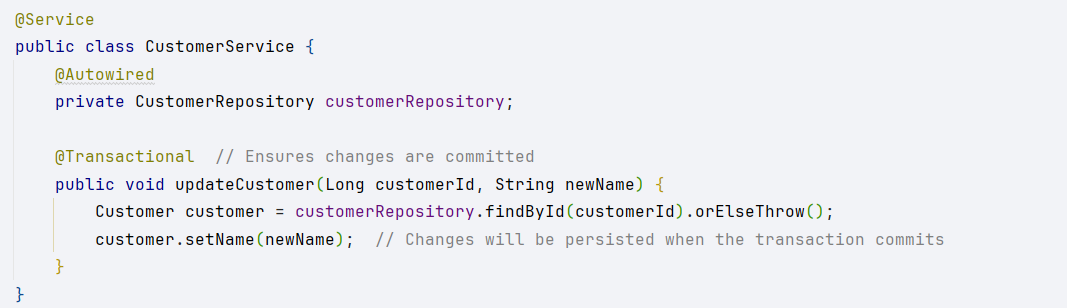
Another common mistake in Hibernate is modifying entities without explicit transactions, leading to unexpected behavior, inconsistent state, or even data loss.

*Example:*

Bad:



Good:

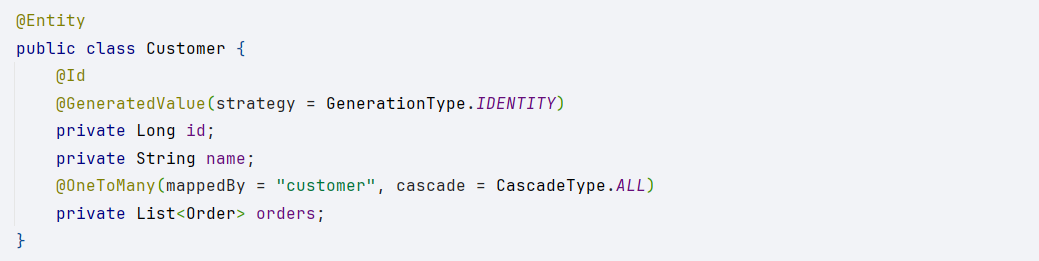


## Blindly Using CascadeType.ALL

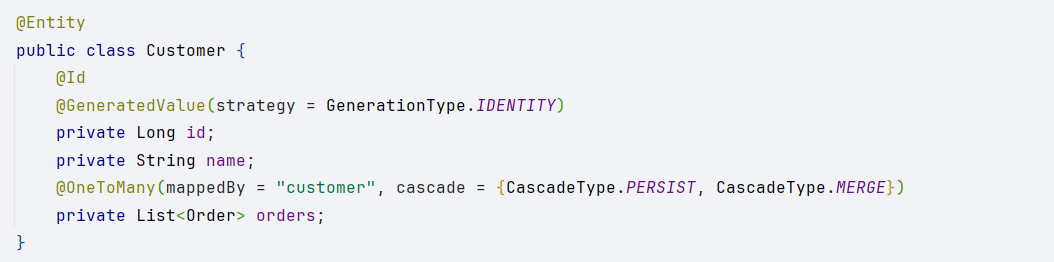
Hibernate provides cascading operations that determine how changes in one entity affect its related entities. One of the most misused cascade types is CascadeType.ALL, which applies ALL operations (PERSIST, MERGE, REMOVE, REFRESH, DETACH) to child entities whenever the parent entity is modified.

*Example:*

Bad:



Good:

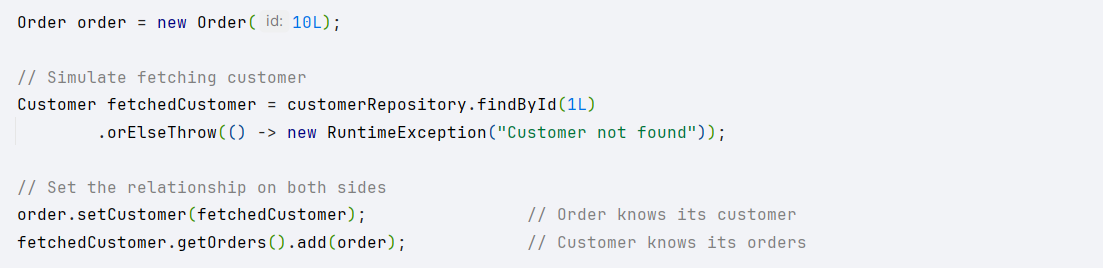


## Manually Updating Both Sides of a Bidirectional Relationship

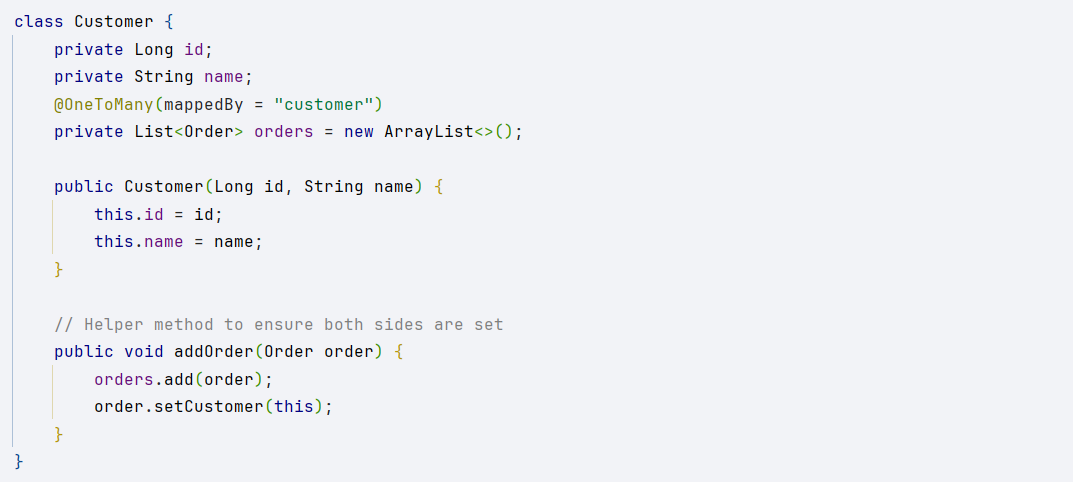
When working with bidirectional relationships, updating both sides manually can lead to inconsistencies.

*Example:*

Bad:



Good:

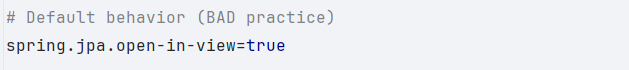


## Leaving EntityManager Open in the Web Layer

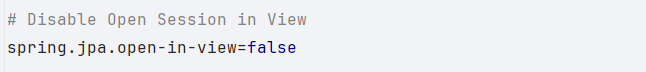
By default, Spring Boot enables Open Session in View (OSIV), which keeps the Hibernate Session (EntityManager) open until the web request completes

*Example:*

Bad:



Good:



## Fetching Large Data Sets Without Pagination

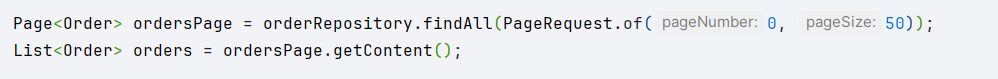
Fetching a large number of records at once without using pagination, leading to high memory consumption and slow performance. Use pagination to fetch data in smaller chunks.

*Example:*

Bad:



Good:

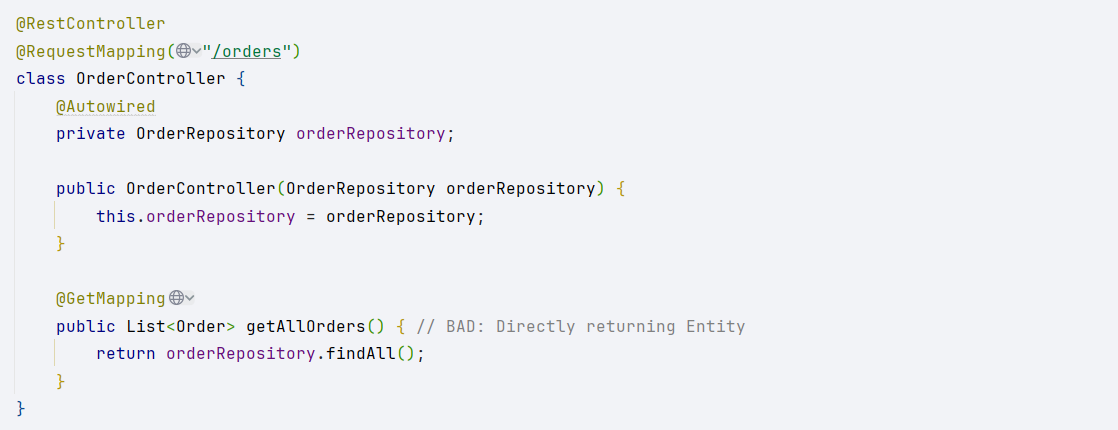


## Exposing Entity Objects Directly in API Responses

Bad Practice: Directly exposing Entity objects in API responses instead of using DTOs.

*Example:*

Bad:



Good:



<https://medium.com/javarevisited/common-hibernate-spring-data-jpa-mistakes-and-how-to-avoid-them-dbc4cd81df71>