Java Development - Best Practices and Guidelines

Version Draft 1.0

Sarvatra Technologies Pvt. Ltd.
2023

Table of Contents

Introduction

These Development Guidelines and coding standards are designed to ensure consistency, readability, and maintainability in Java code. Adhering to these standards will make codebase management more straightforward and enhance collaboration among developers.

Naming Conventions

Choosing appropriate and consistent names for variables, methods, classes, packages and other elements are crucial for code readability and maintainability. The following best practices and guidelines outline recommended practices for naming conventions:

- Use meaningful and descriptive names for variables, methods, classes, and packages.
- Follow the camelCase convention for variables and method names.
- Use PascalCase for class names.
- Constants should be in uppercase with underscores separating words.

Descriptive and Meaningful Names

- Use names that clearly convey the purpose and intent of the variable, method, or class.
- Avoid single-letter variable names (except for loop counters in a **short scope**).
- Choose words that make the code self-documenting.

```
// Good: Descriptive variable name
int numberOfItems;

// Avoid: Non-descriptive variable name
int n;
```

Avoid Acronyms and Abbreviations

- Avoid using acronyms or abbreviations unless they are widely accepted and well-known.
- Prefer full and descriptive words over acronyms or abbreviations.
- If an acronym is commonly accepted (e.g., URL, XML, HTTP), use it consistently.
- Allowed Abbreviations `tl`, `otl`, `x`, `y`, `z`, `jx`, `jy`, `msg`, `sp`, `rrn`, `txnId`, `ctx`, `tid`, `mid`, `sid`, `cfg`, `evt`, `sb` `ss`, `ds`.
- List of Abbreviations to avoid. Use proper names.

Avoid	Use
`acc`	`account`
`mgr`	`manager`
`resp`	`response`
`req`	`request`
`serv`	`servlet`
`hdr`	`header`
`amt`	`amount`
`dt`	`date`
`tm`	`time`

Code :

Test

```
// Good: Descriptive name without acronyms
long employeeId;

// Avoid: Unclear acronym
long empId;
```

```
// Good : well known Abbreviation
package com.sarvatra.xml;

// Avoid
package com.sarvatra.extensibleMarkupLanguage;
```

Consistent and Predictable Naming

- Maintain consistency in naming across the codebase.
- Follow a predictable pattern for similar entities.

```

// Good: Consistent naming
int calculateTotalAmount();
int retrieveTotalAmount();

// Avoid: Inconsistent naming
int calculateAmount();
int getTotal();
```

Follow Project and Industry Conventions

- Adhere to any project-specific or industry-wide naming conventions.
- Ensure consistency with established standards within the development team.

```

// Good: Following project conventions
String databaseUrl;
int maxRetryCount;

// Avoid: Not following project conventions
String dbUrl;
int maxRetries;
```

Avoid Misleading Names

- Choose names that accurately represent the purpose of the entity.
- Avoid names that may mislead developers.

```

// Good: Descriptive and accurate variable name
boolean isUserAuthenticated;

// Avoid: Potentially misleading variable name
boolean validateUser;
```

Variables and Methods

1. **Camel Case:**
   - Follow the camelCase convention for variables and methods.
   - Use camel case for variable names. Start with a lowercase letter, and capitalize the first letter of each subsequent concatenated word.

```
// Good
int studentCount;
```

2. **Descriptive and Meaningful:**
   - Choose variable names that are descriptive and convey the purpose of the variable.

```
// Good
String userName;
```

3. **Avoid Single Letters:**
   - Avoid using single letters for variable names unless the variable represents a loop index.

```
// Good (loop index)
for (int i = 0; i < 10; i++) {
    // ...
}

// Avoid
int x;
```

4. **Avoid Underscores:**
   - Do not use underscores to separate words in variable names. Stick to camel case.

```
// Good
double totalPrice;

// Avoid
double total_price;
```

```
// Good: CamelCase for variables and methods
int calculateTotalAmount() {
    // method implementation
}

// Avoid: Incorrect capitalization
int calculate_total_amount();
```

5. **Start with a Lowercase Letter:**
   - Variable names should start with a lowercase letter.

```
// Good
int itemCount;

// Avoid
int ItemCount;
```

6. **Constants in Uppercase:**

- Constants (final variables) should be in uppercase letters with underscores separating words.

```
// Good
final int MAX_VALUE = 100;

// Avoid
final int maxValue = 100;
```

7. **Use Nouns for Object Instances:**
   - Object instances (e.g., instances of classes) should have names that represent nouns.

```
// Good
Car myCar = new Car();

// Avoid
Car startEngine = new Car();
```

8. **Use Verbs for Methods:**
   - Method names should represent actions and use verbs.
   - Choose method names that represent actions using a verb followed by a noun.

```
// Good
void calculateTotal();

// Avoid
void totalCalculation();
```

```

// Good: Verb-noun naming for methods
void calculateTotalAmount();
void validateUserInput();

// Avoid: Non-descriptive method name
void doSomething();
```

9. **Abbreviations:**
   - Avoid using abbreviations unless they are well-known and widely accepted.

```
// Good
String customerName;

// Avoid
String custName;
```

10. **Plural for Collections:**
    - Use plural names for collections or arrays to indicate that they represent multiple elements.

```
// Good
List<String> students;

// Avoid
List<String> studentList;
```

11. **Consistent Naming Across Scope:**

- Maintain consistency in variable naming across different scopes, such as class level, method level, and loop indices.

```
// Class level
private int totalItems;

// Method level
public void calculateTotalItems() {
    // Local variable
    int itemCount = 0;

    // ...
}
```

12. **Be Mindful of Variable Scope:**
    - Use meaningful names that reflect the variable's scope. For example, prefix class-level variables with "this."

```
// Class level
private int totalItems;

// Method level
public void calculateTotalItems() {
    // Local variable
    int itemCount = 0;

    // Class-level variable
    this.totalItems = itemCount;

    // ...
}
```

13. **Avoid Hiding variable Scope**
    - Try not to add method variables with same names of class variables.
14. **Use Method Overloading cautiously**
    - Use method overloading only if you are adding variant to existing implementation.
    - Overloading is useful to avoid changing all callers if we require to add new functionality.

```

public void updateCustomerStatus(Customer customer, Status status) {
    // logic
}
```

After change:

```
    public void updateCustomerStatus(Customer customer, Status status) {
    // Pass default value, we dont want to change all callers
    this.updateCustomerStatus(customer, status, false);
}

    public void updateCustomerStatus(Customer customer, Status status, boolean notify) {
    // Call from existing function and callers where we want "non-default" ```notify``` value
    // logic
}
```

- Use clear and concise names for methods that describe their actions.

- Follow the JavaBeans standard for getter and setter methods.
  Example: `getUserName()`, `setTotalAmount()`.

Package

Package helps to organize and structure code in a standardized way, making it more readable, maintainable, and scalable.

1. **Lowercase Letters:**
   - Package names should be in all lowercase letters. This enhances readability and consistency.

```
// Good
package com.sarvatra.myproject;

// Avoid
package com.sarvatra.MyProject;
```

2. **Unique and Descriptive:**
   - Package names should be unique and descriptive, providing meaningful information about the contents of the package.

```
// Good
package com.sarvatra.payment;
```

3. **Avoid Single Character Names:**
   - Avoid using single-character names for packages. Use descriptive names that reflect the purpose or content of the package.

```
// Good
package com.sarvatra.utilities;

// Avoid
package com.sarvatra.u;
```

4. **Avoid Java Keywords:**
   - Avoid using Java keywords as package names to prevent potential naming conflicts.

```
// Good
package com.sarvatra.application;

// Avoid
package com.sarvatra.class;
```

5. **Use of Subpackages:**
   - Organize related packages into subpackages when it makes sense. For example, if you have a package for utilities, you might have subpackages like `com.sarvatra.utilities.logging` and `com.sarvatra.utilities.io`.

```
package com.sarvatra.utilities.logging;
```

6. **No Underscores or Hyphens:**
   - Package names should not contain underscores or hyphens. Use camelCase or capitalize separate words.

```
// Good
package com.sarvatra.utilityLibrary;

// Avoid
package com.sarvatra.utility_library;
```

7. **Short and Meaningful:**

- Keep package names reasonably short and meaningful. Long and overly complex package names can be challenging to read and maintain.

```
// Good
package com.sarvatra.data;

// Avoid
package com.sarvatra.models.and.data.processing;
```

9. **Consistent Naming Across Projects:**
   - Maintain consistency in naming conventions across different projects or modules within your organization.

```
// Project 1
package com.sarvatra.project1;

// Project 2
package com.sarvatra.project2;
```

Following these conventions helps create a standardized and organized codebase, making it easier for developers to collaborate, understand, and maintain the code over time.

10. **Suggestions**

Format	Purpose	Example
`com.sarvatra.product`	Product Specific Code	`com.sarvatra.upi` `com.sarvatra.rtsp` `com.sarvatra.imps`
`com.sarvatra.product.bank`	Product Specific Code for Bank	`com.sarvatra.upi.icici` `com.sarvatra.rtsp.icici` `com.sarvatra.imps.icici`

Class

- Use PascalCase (TitleCase) for class names, starting with an uppercase letter.
- Choose nouns or noun phrases for class names.

```
// Good: PascalCase for class names
public class ShoppingCart {
    // class implementation
}

// Avoid: Incorrect capitalization
public class shoppingCart {
    // class implementation
}
```

- One exception to this Rule is Naming `jPOS` participant, `SMPD` activity. As participant and activity refers to some action, we can name it starting with verb.

```

public final class FindOriginal extends SrvtTxnSupport implements HostConstants {
    //
}

//
```

```
public class CheckTransactionStatus extends SrvtTxnSupport implements HostConstants {
    //
}
```

Constants

- Constants should be in uppercase with underscores separating words.
- Use descriptive names for constants.

```
// Good: Uppercase with underscores for constants
public static final int MAX_RETRY_COUNT = 3;

// Avoid: Non-descriptive constant name
public static final int MAX = 3;
```

Regularly Review and Update

- Periodically review and update naming conventions as the codebase evolves.
- Ensure that new additions adhere to established naming guidelines.

```

// Good: Updated naming reflecting changes
double calculateTotalPrice();

// Avoid: Outdated or inconsistent naming
double computeTotal();
```

Consistent and meaningful naming conventions contribute to code clarity and make it easier for developers to understand and maintain the codebase. Regularly communicate and reinforce these conventions within the development team to ensure widespread adoption.

> Avoid giving too big names :)


alt text

Formatting

- Properly format and align code to improve readability.
- Code formatting is a crucial aspect of writing clean, readable, and maintainable Java code. Consistent formatting improves code comprehension, reduces errors, and facilitates collaboration within development teams. Here are some general Java code formatting tips:
- **Intellij Idea** Open Settings > Version Control > Commit. Select on the bottom right under "Commit Check" the `Reformat code`, `Rearrange code`, `Optimize imports` and `Perform Sonarlint analysis` checkboxes.
-

  You can also reformat single file using `CTRL + ALT + SHIFT + L`. You can reformat file whenever possible.
- 
alt text

1. **Indentation:**

- Use consistent indentation to visually structure code blocks.
- Typically, use 4 spaces for each level of indentation.
- Configure your IDE or use a tool like Checkstyle to enforce indentation rules.

2. **Braces Placement:**

- Place opening braces on the same line as the statement or declaration.
- Use braces for control structures, even if they consist of a single statement.
- Use a new line for the opening brace if it's a method or control flow statement (if, for, while).

- Example:

```
if (condition) {
    // code
} else {
    // code
}
```

3. **Line Length:**

- Limit line length to improve readability, usually between 80 and 120 characters.
- Break long lines into multiple lines or use proper indentation to avoid horizontal scrolling.

4. **Spacing:**

- Use consistent spacing around operators (e.g., `=`, `+`, `-`, `==`) to enhance readability.
- Add a space after commas in method parameters and argument lists.
- Example:

```
int result = add(3, 4);
```

5. **Blank Lines:**

- Use blank lines to separate logically related code blocks, methods, or classes.
- Don't use excessive blank lines; aim for a balance to improve code organization.

6. **Comments:**

- Write clear and concise comments to explain complex logic or provide context.
- Avoid unnecessary comments; strive for self-explanatory code.
- Keep comments up-to-date with code changes.

8. **Imports:**

- Organize imports to group them logically.
- Avoid using wildcard imports (`import java.util.*`); explicitly import only the classes needed.
- Use your IDE's feature to remove unused imports.

9. **Consistent Code Structure:**

- Maintain a consistent code structure within your project.
- Enforce coding standards and conventions through code reviews and automated tools.

10. **Version Control Integration:**

- Before committing code changes, ensure that the code adheres to formatting standards.
- Configure pre-commit hooks or use tools like Checkstyle or SpotBugs to enforce coding standards.

11. **Avoid Nested Blocks for Single Statements:**

- While it's syntactically allowed to omit braces for single statements in control structures, it's better to use braces for better readability and to avoid potential bugs.
- Example:

```
// Good
if (condition) {
    // single statement
}

// Avoid
if (condition)
    // single statement
```

12. **Align Code Elements:**

- Align related code elements vertically to enhance readability.
- For example, align variable assignments or method parameters.

These general code formatting tips can significantly contribute to maintaining a clean and consistent codebase. It's essential to establish and follow coding standards within your development team or project and leverage tools and IDE features to enforce these standards. Regular code reviews can also help ensure adherence to formatting guidelines.

Documentation and Comments

- Write clear and concise comments.
- Provide Javadoc comments for classes, methods, and fields.
- Clearly document the purpose, parameters, and return values of methods.
- Use comments to explain complex logic, algorithms, or important decisions.
- **Avoid unnecessary comments;** write self-explanatory code whenever possible.
- For documentation comments, IntelliJ IDEA provides completion that is enabled by default. Type `/**` before a declaration and press Enter . The IDE auto-completes the doc comment for you.
- Keep documentation up-to-date with code changes.
- If possible, generate ASCII/Unicode table and put it in comments if you have decisions/actions based on multiple fields. refer: https://ozh.github.io/ascii-tables/. Following is the Unicode table.

Exception Handling

- Handle exceptions appropriately; do not ignore or swallow them without reason.
- Log exceptions with sufficient information for debugging.
- Favor specific exception types over generic ones.
- Exception handling is a critical aspect of writing robust and reliable Java applications. Well-designed exception handling contributes to code clarity, maintainability, and resilience. Here are best practices for exception handling in Java:

1. Use Specific Exception Types

Catch specific exception types rather than using generic `Exception` classes. This allows for targeted handling and avoids catching unexpected exceptions.

```
try {
    // code that may throw specific exception
} catch (IOException e) {
    // handle IOException
} catch (SQLException e) {
    // handle SQLException
```

2. Avoid Catching Throwable

Avoid catching `Throwable` unless absolutely necessary. Catching `Throwable` may include errors that are not meant to be caught (e.g., `OutOfMemoryError`).

```
// Avoid
try {
    // code
} catch (Throwable t) {
    // handle all throwables
```

3. Fail Fast and Log

If an exception occurs and the application cannot recover, fail fast by letting the exception propagate. Log exceptions with sufficient information for debugging. Include relevant details such as error messages, stack traces, and context.

```
try {
    // code that may throw an exception
} catch (Exception e) {
```

```
    logger.error("An error occurred: {}", e.getMessage());
    throw new CustomException("Failed to process", e);
```

4. Handle Exceptions Appropriately

Handle exceptions at an appropriate level in the application. Consider whether an exception should be handled locally or propagated up the call stack.

```
// Handle locally if possible
try {
    // code that may throw an exception
} catch (FileNotFoundException e) {
    // handle locally
}

// Or propagate up if necessary
throw new CustomException("Failed to process");
```

5. Avoid Empty Catch Blocks

Avoid empty `catch` blocks as they make it difficult to diagnose and troubleshoot issues. Log exceptions or provide meaningful handling within the `catch` block.

```
// Avoid
try {
    // code
} catch (Exception e) {
    // empty catch block
```

6. Use Finally Sparingly

Use `finally` blocks sparingly, typically for resource cleanup (e.g., closing a file or releasing database connections). Be cautious when including logic that could throw exceptions within a `finally` block.

```
try {
    // code
} catch (Exception e) {
    // handle exception
} finally {
    // cleanup code
```

7. Custom Exception Classes

Create custom exception classes to represent specific error conditions in your application. Provide meaningful messages and context in custom exceptions.

```
public class CustomException extends RuntimeException {
    public CustomException(String message) {
        super(message);
```

8. Use Checked Exceptions Appropriately

Prefer unchecked exceptions (those that extend `RuntimeException`) for exceptional conditions that are beyond the control of the calling code. Use checked exceptions sparingly and only for conditions that the calling code can reasonably be expected to handle.

```
// Unchecked exception
public class CustomRuntimeException extends RuntimeException {
```

```
    public CustomRuntimeException(String message) {
        super(message);
```

9. Exception Translation

Consider translating exceptions between layers of your application to maintain a clear separation of concerns. Convert lower-level exceptions into higher-level exceptions that are more appropriate for the higher-level context.

```
public void businessMethod() throws BusinessException {
    try {
        // lower-level code
    } catch (LowerLevelException e) {
        throw new BusinessException("Error in business logic", e);
```

10. Use Java 7+ Try-With-Resources

When working with resources such as files or database connections, use the try-with-resources statement to automatically close them.

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    // code that uses br
```

11. Avoid Swallowing Exceptions

Avoid situations where exceptions are caught and ignored without appropriate logging or handling. If an exception cannot be handled, consider logging it for future diagnosis.

```
try {
    // code that may throw an exception
} catch (Exception e) {
    logger.warn("An exception occurred but was ignored: {}", e.getMessage());
```

12. Use Logging Frameworks

- Utilize logging frameworks like SLF4J to log exceptions. Log enough information to diagnose the issue but avoid leaking sensitive information.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

private static final Logger logger = LoggerFactory.getLogger(MyClass.class);

try {
    // code
} catch (Exception e) {
    logger.error("An error occurred: {}", e.getMessage());
```

- For jPOS application, use LogEvent and log it as "ERROR".
- Avoid `printStackTrace`.
- Avoid `System.out.println`
- Avoid `System.err.println`

Class Structure

- Organize classes and packages logically.
- Follow the Single Responsibility Principle (SRP) and keep classes focused on a single concern.
- Use appropriate access modifiers for fields and methods.

Designing a well-structured class is crucial for writing maintainable and scalable Java code. Here are some best practices for class structure in Java:

1. **Single Responsibility Principle (SRP):**

- Each class should have a single responsibility or reason to change. This promotes maintainability and makes the code easier to understand.

2. **Encapsulation:**

- Encapsulate the internal state of a class and provide controlled access through methods (getters and setters). This helps in data hiding and reduces coupling.

3. **Consistent Naming Conventions:**

- Follow consistent and meaningful naming conventions for classes, methods, and variables. Use CamelCase for class names and methods, and use lowercase for variables.

4. **Use of Access Modifiers:**

- Always define access modifiers for classes, variables, methods.
    - Default i.e. "package private" access for variables, methods should be used only if it's required to visible within package. Otherwise, `private` access modifier should be used.
    - Classes should be always Default to "package private".
    - Avoid making class variables as protected, use `protected` setters and getters. If field is `protected` then it shall be `final`.
    - `public` and `protected` methods should always mark as `final` unless overriding is expected.
    - Constant fields should be always `static final` irrespective of access modifiers.

- Clearly define the access modifiers for class members (fields, methods). Keep fields private and provide necessary access through public methods.
    - **Private Access:** Use private access for fields and methods that should only be accessible within the declaring class. This enforces encapsulation and limits direct access from outside classes. **while writing code, we should start with providing `private` access.
    - **Default (Package-Private) Access:** Use default (package-private) access for classes, methods, and fields when they should only be accessible within the same package. This is the default access level if no modifier is specified.
    - **Protected Access:** Use protected access for methods and fields that should be accessible within the same package and by subclasses. Be cautious with the use of protected members to avoid unnecessary coupling.
    - **Public Access:** Use public access for methods and fields that are part of the class's public API. This allows access from any class and is appropriate for methods that need to be accessible globally.
    - **Minimize Access Levels:** Minimize the visibility of classes, methods, and fields. Use the most restrictive access level that still allows the functionality needed. This helps control the scope of access and reduces coupling.
    - **Encapsulation:** Encapsulate internal state by making fields private or protected and providing controlled access through public methods (getters and setters). This promotes data hiding and modular design.
    - **Avoid Public Fields:** Avoid using public fields. Instead, use private fields with public accessor methods (getters and setters) to control access and modification. This provides flexibility for future changes.
    - **Immutable Classes:** Consider making classes immutable by using private fields and not providing setters. If modification is necessary, create a new instance with the modified values.
    - **Use Access Modifiers Consistently:** Be consistent in using access modifiers throughout the codebase. Follow a coding convention that specifies when to use each access level.
    - **Package-Level API:** When designing packages, consider whether certain classes or methods should be part of the package's API. If so, use public or protected access accordingly.
    - **Final Classes and Methods:** Consider marking classes and methods as `final` when they are not meant to be extended or overridden. This can help ensure that the intended behavior is maintained.
    - **Avoid Excessive Use of Public Members:** Limit the number of public methods and fields in a class. Too many public members can make the class harder to understand and maintain.
    - **Effective Use of Interfaces:** When defining interfaces, prefer default or private methods over public methods to avoid unnecessary exposure. Only expose methods in the interface that are part of the intended public API.
    - **Access Modifier for Top-Level Classes:** Use the appropriate access modifier for top-level classes. If a class is intended to be used outside the package, make it public. Otherwise, use default access.
    - **Consider Future Changes:** Anticipate potential changes in your codebase. If there's a possibility of a class or method needing wider access in the future, choose access modifiers accordingly.
    - **Document Access Levels:** Use comments or documentation to explain the rationale behind the chosen access levels, especially for public and protected members. Describe how they are intended to be used.

      By following these best practices, you can ensure that your use of access modifiers aligns with principles of encapsulation, maintainability, and flexibility in your Java code. Always consider the specific requirements of your application and adapt these practices accordingly.

5. **Immutable Classes:**

- Consider making classes immutable when possible. Immutable classes are thread-safe and have predictable behavior.

6. **Composition over Inheritance:**

- Prefer composition over inheritance to achieve code reuse. Composition is often more flexible and leads to less coupled code.

7. **Default Constructor:**

- Provide a default constructor only if needed. If other constructors are defined, and no default constructor is provided, Java will not automatically generate one.

8. **Proper Use of Static Members:**

- Use static members (methods or fields) judiciously. Static methods should be stateless and perform operations that are not dependent on instance-specific data.

9. **Well-Defined Relationships:**

- Clearly define relationships between classes, such as associations, aggregations, and compositions. Use appropriate UML diagrams or annotations to document these relationships.

10. **Effective Use of Interfaces:**

- Utilize interfaces to define contracts and promote loose coupling. Follow the interface segregation principle and create cohesive interfaces.

11. **Serializable Classes:**

- Implement the `Serializable` interface carefully. If a class is not meant to be serialized, consider using the `transient` keyword for fields that should not be serialized.

12. **Effective Use of Enumerations:**

- Use enumerations for representing a fixed set of constants or options. Enumerations can enhance code readability and reduce the chances of invalid values.

13. **Logging:**

- Use logging frameworks like SLF4J to log important events and errors. Avoid using `System.out.println()` for logging in production code.

14. **JavaDoc Comments:**

- Include meaningful JavaDoc comments to document the purpose, behavior, and usage of classes and methods. This aids in code understanding and documentation generation.

15. **Consistent Formatting:**

- Adopt a consistent code formatting style. Use indentation, braces placement, and line breaks consistently across your codebase.

16. **Testing Considerations:**

- Design classes to be testable. Consider using dependency injection to facilitate unit testing, and adhere to the principles of test-driven development (TDD).

17. **Effective Exception Handling:**

- Implement robust exception handling. Use checked exceptions for recoverable errors and runtime exceptions for unexpected errors.

18. **Version Control Integration:**

- Ensure that your class files are version-controlled. Use appropriate versioning strategies and commit messages for tracking changes.

19. **Code Review:**

- Regularly conduct code reviews to ensure that class structures align with coding standards and best practices. Code reviews can catch potential issues early in the development process.

20. **Continuous Refactoring:**

- Embrace continuous refactoring. As the codebase evolves, refactor classes to maintain code quality and adhere to changing requirements.

By following these best practices, you can create well-organized and maintainable class structures in your Java applications. Keep in mind that these principles are guidelines, and their application might vary based on specific project requirements and constraints.

Best Practices and things to avoid (*Don'ts*)

- Follow the Java coding conventions outlined in the official Oracle Java Code Conventions.
- Keep methods short and focused on a specific task.
- Use version control effectively and make meaningful commit messages.
- Regularly refactor and optimize code to improve performance.

Feel free to customize these standards based on project-specific requirements and keep this document updated as coding practices evolve.

It seems like there might be a typo or a missing context in your question. If you are asking about something related to Java enterprise applications and using the word "Dont," it would be helpful if you could provide more details or clarify your question.

1. **Don't Hardcode Values:** Avoid hardcoding values like database connection strings, API endpoints, or configuration parameters directly in your code. Use configuration files or external configuration mechanisms.
2. **Don't Ignore Exception Handling:** Proper exception handling is crucial in enterprise applications. Catch exceptions at the appropriate level and handle them gracefully, providing meaningful error messages and logging.
3. **Don't Neglect Security:** Pay attention to security best practices. Sanitize inputs, use parameterized queries to prevent SQL injection, and follow secure coding guidelines.
4. **Don't Use Deprecated APIs:** Stay updated with the latest Java versions and libraries. Avoid using deprecated APIs, as they may be removed in future versions.
5. **Don't Overlook Performance:** Optimize your code for performance. Avoid unnecessary database queries, use caching where appropriate, and profile your application to identify bottlenecks.
6. **Don't Forget Logging:** Logging is essential for troubleshooting and monitoring. Implement proper logging throughout your application, and log relevant information at different levels.
7. **Don't Use Blocking Operations in the Main Thread:** In a Java enterprise application, avoid performing blocking operations (like I/O operations) in the main thread, as it can lead to poor responsiveness. Use asynchronous or background processing where needed.
8. **Don't Ignore Code Quality:** Follow coding standards, use meaningful variable and method names, and consider code readability. Regularly review and refactor your code to maintain a high level of quality.
9. **Don't Reinvent the Wheel:** Leverage existing libraries and frameworks when appropriate. Reusing well-established components can save development time and improve the maintainability of your code.
10. **Don't Forget Testing:** Implement comprehensive testing, including unit tests, integration tests, and end-to-end tests. Automated testing helps ensure the reliability and correctness of your application.
11. **Don't Forget Scalability:** Design your application with scalability in mind. Consider potential future growth and design your architecture to handle increased loads. Use scalable databases, caching mechanisms, and distributed systems where necessary.
12. **Don't Ignore Transactions:** If your application involves database operations, use transactions appropriately. Ensure that your transactions are ACID-compliant to maintain data consistency.
13. **Don't Use String Concatenation in SQL Queries:** Avoid constructing SQL queries by concatenating strings, as it makes your application vulnerable to SQL injection attacks. Instead, use parameterized queries or prepared statements.
14. **Don't Neglect Connection Pooling:** Use connection pooling to manage database connections efficiently. Creating and closing connections for every database operation can lead to performance issues.
15. **Don't Hardcode Credentials:** Never hardcode sensitive information like usernames, passwords, or API keys in your source code. Utilize secure credential storage mechanisms or environment variables.
16. **Don't Overuse Eager Loading:** Be cautious with eager loading in ORM (Object-Relational Mapping) frameworks. Loading too much data eagerly can impact performance. Use lazy loading where appropriate to fetch data only when needed.

17. **Don't Rely Solely on Framework Defaults:** While Java enterprise frameworks provide defaults and conventions, be sure to understand and configure them according to your application's specific needs. Don't assume that the defaults are always the best fit.

18. **Don't Forget Monitoring and Metrics:** Implement proper monitoring and metrics in your application. Use tools like JMX, Prometheus, or other monitoring solutions to gain insights into the health and performance of your application.

19. **Don't Neglect Documentation:** Document your code, APIs, and architectural decisions. Clear documentation makes it easier for other developers (and your future self) to understand and maintain the codebase.

20. **Don't Skip Code Reviews:** Conduct regular code reviews to ensure code quality, adherence to coding standards, and the identification of potential issues early in the development process.

21. **Don't Use Blocking I/O in Servlets:** In web applications, avoid performing blocking I/O operations in servlets, as it can lead to poor application responsiveness. Use asynchronous servlets or offload blocking operations to background threads.

22. **Don't Forget Internationalization:** If your application is intended for a global audience, consider internationalization (i18n) from the beginning. Externalize strings and support multiple languages to make your application more accessible.

23. **Don't use System.exit:**

24. **Do Follow Coding Standards:**
    - Adhere to established coding standards and conventions for consistency.
    - Follow the Java Code Conventions and use tools like Checkstyle or SonarQube to enforce them.

25. **Do Use Descriptive Naming:**
    - Choose meaningful and descriptive names for classes, methods, and variables.
    - Make your code self-documenting by using clear and expressive names.

26. **Do Practice Modularization:**
    - Break down your code into small, modular components with well-defined responsibilities.
    - Follow the principles of modularity and encapsulation for maintainability.

27. **Do Version Control:**
    - Use version control systems (e.g., Git) to track changes and collaborate effectively.
    - Commit frequently, write meaningful commit messages, and leverage branching for feature development.

28. **Do Exception Handling:**
    - Handle exceptions gracefully and appropriately for the context.
    - Use try-catch blocks for exception handling, and log exceptions for troubleshooting.

29. **Do Use Design Patterns:**
    - Familiarize yourself with common design patterns and apply them where appropriate.
    - Design patterns provide proven solutions to recurring design problems.

30. **Do Optimize Performance:**
    - Profile your code to identify performance bottlenecks.
    - Optimize critical sections, use efficient algorithms, and consider caching strategies.

31. **Do Write Unit Tests:**
    - Practice test-driven development (TDD) by writing unit tests before implementing functionality.
    - Use testing frameworks like JUnit for automated testing.

32. **Do Document Your Code:**
    - Provide clear and concise documentation for classes, methods, and important code blocks.
    - Use Javadoc to document public APIs.

33. **Do Use Dependency Injection:**
    - Apply the principle of dependency injection for better testability and flexibility.
    - Use frameworks like Spring for managing dependencies.

34. **Do Continuous Integration:**
    - Set up a continuous integration (CI) pipeline to automate build and testing processes.
    - Integrate tools like Jenkins, Travis CI, or GitLab CI into your development workflow.
35. **Do Secure Coding:**
    - Follow secure coding practices to prevent vulnerabilities.
    - Validate inputs, avoid hardcoded credentials, and use encryption where necessary.
36. **Do Regular Code Reviews:**
    - Conduct regular code reviews to catch issues early and ensure code quality.
    - Provide constructive feedback and foster a collaborative development environment.
37. **Do Use Thread-Safe Practices:**
    - Be aware of multithreading issues and use thread-safe data structures and synchronization where needed.
    - Follow best practices for concurrent programming.
38. **Do Optimize Memory Usage:**
    - Be mindful of memory consumption and avoid memory leaks.
    - Use tools like VisualVM or Eclipse Memory Analyzer for profiling memory usage.
39. **Do Plan for Internationalization:**
    - If your application is intended for a global audience, plan for internationalization (i18n) from the start.
    - Externalize and manage strings for easy translation.
40. **Do Consider Microservices Architecture:**
    - Evaluate the use of microservices architecture for scalability and maintainability.
    - Design services with clear boundaries and communication mechanisms.
41. **Do Keep Libraries and Dependencies Updated:**
    - Regularly update libraries and dependencies to benefit from bug fixes and improvements.
    - Use tools like Maven or Gradle to manage dependencies.
42. **Do Use Interfaces and Abstraction:**
    - Program to interfaces rather than concrete implementations.
    - Use abstraction to decouple components and improve flexibility.
43. **Do Plan for Continuous Improvement:**
    - Embrace a mindset of continuous improvement.
    - Learn from experiences, adopt new technologies, and evolve your development practices.

Building a payment application requires careful consideration of security, reliability, and adherence to industry standards. Here are some best practices for developing payment application:

1. **Security Best Practices:**
    - **Encryption:** Use strong encryption algorithms for sensitive data, including cardholder information, during transmission and storage.
    - **Tokenization:** Implement tokenization to replace sensitive data with unique identifiers, reducing the risk associated with storing critical information.
    - **PCI Compliance:** Adhere to Payment Card Industry Data Security Standard (PCI DSS) requirements to ensure the secure handling of payment card data.
2. **Transaction Handling:**
    - **Atomic Transactions:** Design transactions to be atomic, ensuring that they either complete successfully or leave no trace in case of failure.
    - **Transaction Logging:** Implement comprehensive transaction logging to facilitate auditing, troubleshooting, and reconciliation.
3. **Error Handling:**

- **Graceful Error Handling:** Provide clear and informative error messages to users and log detailed error information for support and debugging purposes.
- **Fallback Mechanism:** Implement a fallback mechanism to handle transaction failures and ensure a consistent user experience.

4. **Concurrency and Scalability:**
   - **Thread Safety:** Ensure thread safety in your application, especially when dealing with shared resources.
   - **Scalability:** Design your application to scale horizontally to handle increased transaction loads.
5. **Configuration Management:**
   - **Externalize Configuration:** Externalize configuration parameters such as endpoints, timeout values, and connection details to make adjustments without code changes.
   - **Secure Configuration:** Store sensitive configuration information securely, using encryption or other appropriate measures.
6. **Logging and Monitoring:**
   - **Comprehensive Logging:** Implement thorough logging of application events, errors, and transactions for monitoring and auditing purposes.
   - **Monitoring Tools:** Integrate monitoring tools to track system performance, detect anomalies, and proactively address issues.
7. **Code Quality and Testing:**
   - **Code Reviews:** Conduct regular code reviews to ensure adherence to coding standards, security practices, and best coding practices.
   - **Automated Testing:** Develop a robust suite of automated tests, including unit tests, integration tests, and end-to-end tests, to ensure the reliability of your application.
8. **Documentation:**
   - **Code Documentation:** Provide detailed documentation for your code, APIs, and configuration settings to facilitate maintenance and future development.
   - **User Manuals:** Create user manuals and developer documentation to guide users and developers in understanding and using the application.
9. **Compliance with Standards:**
   - **Payment Industry Standards:** Stay current with payment industry standards and regulations to ensure compliance with legal and security requirements.
10. **Integration and Interoperability:**
    - **API Design:** Design clean and well-documented APIs to facilitate integration with other systems.
    - **Interoperability Testing:** Perform thorough interoperability testing with different devices and platforms to ensure seamless integration.
11. **Continuous Monitoring and Updates:**
    - **Security Updates:** Regularly update your application and libraries to address security vulnerabilities promptly.
    - **Continuous Improvement:** Foster a culture of continuous improvement, incorporating feedback and lessons learned into the development process.
12. **Secure Deployment:**
    - **Secure Configuration:** Securely configure the deployment environment, including access controls and network security measures.
    - **Regular Audits:** Conduct regular security audits to identify and address potential vulnerabilities.
13. **Secure Network Communication:**
    - Use secure communication protocols such as TLS/SSL to encrypt data transmitted over networks.
    - Regularly update and patch your systems to address any known vulnerabilities in network protocols.
14. **Secure Cardholder Data Storage:**
    - Minimize the storage of sensitive cardholder data. If storage is necessary, encrypt the data using strong cryptographic algorithms.
    - Implement access controls to restrict access to stored cardholder data only to authorized personnel.
15. **Transaction Reconciliation:**

- Implement robust reconciliation mechanisms to ensure that transactions are accurately recorded and settled.
- Conduct regular reconciliation processes to identify and resolve discrepancies.

16. **Device Management:**
    - Regularly monitor and manage connected devices to ensure they are operational and secure.
    - Implement device health checks and perform periodic firmware updates.
17. **Transaction Integrity:**
    - Design transactions to be idempotent, allowing the same transaction to be safely replayed without unintended side effects.
    - Implement mechanisms to detect and prevent duplicate transactions.
18. **User Authentication and Authorization:**
    - Enforce strong authentication mechanisms for users accessing the application.
    - Implement role-based access controls to restrict users' permissions based on their roles.
19. **Secure Key Management:**
    - Follow best practices for key management, including secure generation, storage, and rotation of cryptographic keys.
    - Utilize Hardware Security Modules (HSMs) for added protection of cryptographic keys.
20. **Comprehensive Testing:**
    - Conduct thorough security testing, including penetration testing and vulnerability assessments, to identify and address potential security vulnerabilities.
    - Perform regular security audits to assess the overall security posture of the application.
21. **Mobile Payment Considerations:**
    - If your payment application includes mobile components, secure mobile communication channels and implement secure storage practices for mobile devices.
    - Implement secure authentication methods for mobile users.
22. **Fraud Detection and Prevention:**
    - Implement fraud detection mechanisms to identify and prevent fraudulent transactions.
    - Regularly update fraud prevention rules based on the evolving nature of fraudulent activities.
23. **Emergency Response Plan:**
    - Develop a comprehensive incident response plan to address security incidents promptly.
    - Conduct regular drills and simulations to ensure that the response team is well-prepared.
24. **Data Masking and Redaction:**
    - Implement data masking and redaction techniques to protect sensitive information displayed in logs, reports, or user interfaces.
    - Minimize the exposure of sensitive data to users and system administrators.
25. **Payment Gateway Integration:**
    - If integrating with a payment gateway, follow best practices provided by the gateway provider.
    - Ensure that the integration adheres to industry standards and complies with relevant regulations.
26. **Continuous Compliance Monitoring:**
    - Regularly monitor changes in regulatory requirements and update your application to remain compliant.
    - Engage with legal and compliance experts to stay informed about changes in the regulatory landscape.
27. **Third-Party Security Assessment:**
    - Perform security assessments of third-party components and services used in your application.
    - Ensure that vendors providing payment-related services adhere to security best practices.
28. **Data Retention Policies:**
    - Define and enforce data retention policies to minimize the storage of unnecessary transaction data.
    - Regularly purge or archive data in accordance with these policies.
29. **Secure Input Validation:**

- Implement rigorous input validation to prevent common security vulnerabilities such as injection attacks (e.g., SQL injection, command injection).
- Validate and sanitize user inputs on both the client and server sides.

30. **Secure Communication with External Systems:**
    - Ensure that communications with external systems, such as payment processors and third-party services, are secure and encrypted.
    - Use secure and mutually authenticated channels when exchanging sensitive information with external entities.
31. **Regular Security Training:**
    - Provide regular security training to development and operational teams to keep them informed about the latest security threats and best practices.
    - Foster a security-aware culture within the organization.
32. **Secure Code Reviews:**
    - Conduct thorough security-focused code reviews to identify and mitigate potential vulnerabilities.
    - Leverage static analysis tools to assist in identifying security issues in the codebase.
33. **Secure Session Management:**
    - Implement secure session management practices, including secure session token generation, storage, and transmission.
    - Use secure, randomly generated session IDs and enforce session timeout policies.
34. **Implement Two-Factor Authentication (2FA):**
    - Consider implementing two-factor authentication for user accounts accessing sensitive functions or settings.
    - 2FA adds an additional layer of security beyond username and password.
35. **Secure File Handling:**
    - Implement secure file handling practices, especially when dealing with configuration files, logs, and other data storage.
    - Restrict access to files containing sensitive information and encrypt sensitive data when stored.
36. **Secure Configuration Management:**
    - Manage configurations securely by encrypting sensitive information, using secure channels for configuration updates, and avoiding hardcoded secrets.
    - Regularly review and update configurations based on security best practices.
37. **Secure Development Life Cycle (SDLC):**
    - Integrate security into the entire development life cycle, from design to deployment.
    - Perform security reviews at each stage of the SDLC to identify and address potential vulnerabilities early.
38. **Data Loss Prevention (DLP):**
    - Implement data loss prevention mechanisms to monitor, detect, and prevent unauthorized access or transmission of sensitive information.
    - Define policies to control the flow of sensitive data within the application.
39. **Dynamic Application Security Testing (DAST):**
    - Conduct dynamic application security testing to simulate real-world attack scenarios.
    - Use DAST tools to identify vulnerabilities in runtime and assess the security of the application in a production-like environment.
40. **Secure Docker Containerization:**
    - If using containerization (e.g., Docker), follow security best practices for securing containerized applications.
    - Regularly update base images, scan images for vulnerabilities, and restrict container privileges.
41. **Secure Code Dependencies:**
    - Regularly audit and update third-party libraries and dependencies to patch known vulnerabilities.
    - Use dependency scanning tools to identify and mitigate security risks associated with libraries.
42. **Incident Response Plan Testing:**
    - Regularly test your incident response plan through simulated exercises and tabletop scenarios.
    - Ensure that the response team is familiar with procedures and can efficiently respond to security incidents.

43. **Secure Remote Access:**
    - If the application allows remote access, secure it using secure protocols (e.g., VPN) and enforce strong authentication.
    - Limit and monitor remote access to prevent unauthorized entry points.
44. **Penetration Testing:**
    - Conduct regular penetration testing to identify and address security weaknesses.
    - Address findings promptly to improve the security posture of the application.
45. **Audit Trails and Forensic Analysis:**
    - Implement comprehensive audit trails for tracking critical events and transactions.
    - Enable forensic analysis capabilities to investigate security incidents and trace the root causes.

Remember that security is an ongoing process, and it's crucial to stay informed about emerging threats and best practices. Regularly review and update security measures to adapt to evolving risks and ensure the continued security of payment application.

Tips

1. All Urls, HTTP client parameters, ip/port should be read from environment OR from database.
    - This way we can easily keep single file for all environment.
    - For jPOS application, we can keep all deployment descriptors in `src/main/resources/META-INF/q2/deploy` and related configuration if required in `src/main/resources/META-INF/q2/cfg`
    - Configure and use Freemarker Decorator from jPOS-EE, with this we can read properties from environment.
    - Use proper jPOS configuration factory.

Configuration Factory	Usage
`org.jpos.q2.SimpleConfigurationFactory`	Default From jPOS library class.
`org.jpos.ee.SysConfigConfigurationFactory`	From jPOS library class.
`org.jpos.util.GenericConfigurationFactory`	From Sarvatra JPTS library (Sarvatra Implementation).

2. Institute specific transaction flow and decisions should be driven from `SysConfig`, `InstituteConfig` or `EntityConfig`.
    - Product Specific `CheckInstitute`, `CheckConsumer` implementation can put such properties in Context. In fact we can have one property in config as `put-context-properties`, iterate and populate these properties in Context.
    - If possible, use existing jPOS provided GroupSelector `org.jpos.transaction.participant.Switch`. You simply need to configure

```
<property name="txnname" value="WHATEVER_PROPERTY_YOU_WANT"/>
```

3. Pausable Participants requires special care. There must be someone (other thread/service) waking up the context! So while writing Pausable participant, ensure context is already registered for implementation for Async callback.
4. Use `DB.exec` or `DB.execWithTransaction` for database call in Service/Helper classes.
5. `SimpleLogListener` should only enabled for development environment.

```
<log-listener class="org.jpos.util.SimpleLogListener" enabled="${DEV_ENVIRONMENT}" />
```

Here if required, `DEV_ENVIRONMENT` can be configured as environment variable.

```
export DEV_ENVIRONMENT=true
```

6. As per product requirements, Configure log-rotations (size/duration) and history.
7. Remember Deploy files will be deployed in ascending alphabetical order NOT in ascending numerical order. So for files `10_test.xml`, `99_test.xml` and `100_test.xml` - deployment order will be

1. `100_test.xml`
2. `10_test.xml`
3. `99_test.xml`

So, number/organize services such as "independent" services will be deployed first. and then services which are dependent on those services will be deployed.

Some service naming conventions,

Purpose	Service
logger	`00_*_logger.xml`
db	`01_*_db.xml`
transaction manager	`10_*_txnmgr.xml`
channel	`10_*_channel.xml`
mux	`20_*_mux.xml`
mux pool	`21_*_muxpool.xml`
logon	`30_*_logonmgr.xml`
qserver	`50_*_server.xml`
jetty	`90_*_jetty.xml`

8. While shutdown, services will be un-deployed in descending alphabetical order.