**1. Ensure the code follows standard naming conventions**

Follow Java code conventions

Following language conventions helps quickly skim through the code and make sense of it, thereby improving readability. For instance, all package names in Java are written in lowercase, constants in all caps, variable names in CamelCase, etc. Find the complete list of conventions here.

Some teams develop their own conventions, so be flexible in such cases!

Meaningful naming conventions help ensure the readability and maintainability of the application.

As such, ensure variable, method, and class names convey the subject:

*addPerson()*

Could be clarified to:

*addEmployee()*

Use all lower cases for package names and use reversed Internet domain naming conventions:

*org/companyname/appname*

Class names should start with Capitals:

*Employee, Student,*

Variable and method names should use CamelCase:

*employeeList, studentName, displayEmployees()*

**2. Make sure it handles constants efficiently**

Constants help improve memory as they are cached by the JVM. For values that are reused across multiple places, create a constant file that holds static values.

Favour database-driven values over dynamic values. Also, use ENUMs to group constants.

**3. Check for proper clean Up**

It is common during development to use procedures that help with your coding and debugging (hard coded variables, for example). It is good practice to remove these items and others such as:

* Console print statements
* Unnecessary comments
* Use @deprecated on method/variable names that aren’t meant for future use

**4. Handle strings appropriately**

If you need to perform a lot of operations on a [String](https://stackabuse.com/string-vs-stringbuilder-vs-stringbuffer-in-java/), use StringBuilder or StringBuffer.

Strings are immutable, whereas [StringBuilder](https://docs.oracle.com/javase/tutorial/java/data/buffers.html) and StringBuffer are mutable and can be changed. Additionally, a new String object is created for every concatenation operation.

Rather than creating multiple items, using a mutable object is preferred.

**5. Optimize to use switch-case over multiple If-Else statements**

Rather than using multiple if-else conditions, use the cleaner and more readable switch-case.

Doing so makes the logic cleaner and optimizes the app's performance.

switch(expression) {

 case x:

// code block

   break;

case y:

  // code block

   break;

 default:

   // code block

}

**6. Ensure the code follows appropriate error handling procedures**

The **NullPointerException**is one of the most common and frustrating exceptions.

However, they can be avoided by performing a null check on a variable before operating on it.

The best practice is to use checked exceptions for recoverable operations and use runtime exceptions for programming errors.

Another area to evaluate during a Java code review is to ensure all exceptions are wrapped in custom exceptions.

In this way, the stack trace is preserved, making it easier to debug when things go wrong.

Also, it should declare specific checked exceptions that the method throws rather than generic ones. Doing so doesn’t give you the option to handle and debug the issue appropriately.

**Avoid this:**

public void hello() throws Exception { //Incorrect way

}

**Try this instead:**

public void hello() throws SpecificException1, SpecificException2 { //Correct way

}

Use the try-catch block for exception handling with appropriate actions taken in the catch block.

Also, use a finally block to release resources, such as database connections, in the finally block. This allows you to close the resource gracefully.

**7. Avoid unnecessary comments in code?**

Comments should not be used to explain code. If the logic is not intuitive, it should be rewritten. Use [comments](https://stackoverflow.blog/2021/07/05/best-practices-for-writing-code-comments/)to answer a question that the code can’t.

Another way to state it is that the comment should explain the “why” versus “what” it does.

**8. Validate that the code follows separation of concerns**

Ensure there is no duplication. Each class or method should be small and focus on one thing.

**For example:**

EmployeeDao.java - Data access logic

Employee.java - Domain Logic

EmployeerService.java - Business Logic

EmployeeValidator.java - Validating Input Fields

**9. Does the code rely on frameworks rather than custom logic when possible?**

When time is of the essence, reinventing the wheel is time wasted. There are plenty of proven frameworks and libraries available for the most common use cases you may need.

Examples include Apache Commons libraries, Spring libraries, and XML/JSON libraries.

**10. Make sure variables don’t cause memory leaks**

Creating a bunch of unnecessary variables can overwhelm the heap and lead to [memory leaks](https://docs.oracle.com/javase/10/troubleshoot/troubleshoot-memory-leaks.htm#JSTGD266) and cause [performance](https://www.adservio.fr/post/bringing-performance-to-java-developers) problems. This occurs when an object is present in the heap but is no longer used, and the garbage collection cannot remove it from memory.

Example:

**Avoid This**

*boolean removed = myItems.remove(item); return removed;*

**Try This Instead**

*return myItems.remove(item);*

11. **Replace imperative code with lambdas and streams**

If you're using Java 8+, replacing loops and extremely verbose methods with streams and lambdas makes the code look cleaner. Lambdas and streams allow you to write functional code in Java. The following snippet filters odd numbers in the traditional imperative way:

List<Integer> oddNumbers = new ArrayList<>();

for (Integer number : Arrays.asList(1, 2, 3, 4, 5, 6)) {

if (number % 2 != 0) {

oddNumbers.add(number);

}

}

This is the functional way of filtering odd numbers:

List<Integer> oddNumbers = Stream.of(1, 2, 3, 4, 5, 6)

.filter(number -> number % 2 != 0)

.collect(Collectors.toList());

12.**Beware of the NullPointerException**

When writing new methods, try to avoid returning nulls if possible. It could lead to null pointer exceptions. In the snippet below, the highest method returns a null if the list has no integers.

class Items {

private final List<Integer> items;

public Items(List<Integer> items) {

this.items = items;

}

public Integer highest() {

if (items.isEmpty()) return null;

Integer highest = null;

for (Integer item : items) {

if (items.indexOf(item) == 0) highest = item;

else highest = highest > item ? highest : item;

}

return highest;

}

}

Before directly calling a method on an object I recommend checking for nulls as shown below.

Items items = new Items(Collections.emptyList());

Integer item = items.highest();

boolean isEven = item % 2 == 0; // throws NullPointerException ❌

boolean isEven = item != null && item % 2 == 0 // ✅

It can be pretty cumbersome to have null checks everywhere in your code though. If you are using Java 8+, consider using the Optional class to represent values that may not have valid states. It allows you to easily define alternate behavior and is useful for chaining methods.

In the snippet below, we are using Java Stream API to find the highest number with a method which returns an Optional. Note that we are using Stream.reduce, which returns an Optional value.

public Optional<Integer> highest() {

return items

.stream()

.reduce((integer, integer2) ->

integer > integer2 ? integer : integer2);

}

Items items = new Items(Collections.emptyList());

items.highest().ifPresent(integer -> { // ✅

boolean isEven = integer % 2 == 0;

});

Alternatively, you could also use annotations such as @Nullable or @NonNull which will result in warnings if there is a null conflict while building the code. For instance, passing a @Nullable argument to a method that accepts @NonNull parameters.

13. **Directly assigning references from client code to a field**

References exposed to the client code can be manipulated even if the field is final. Let's understand this better with an example.

private final List<Integer> items;

public Items(List<Integer> items) {

this.items = items;

}

In the above snippet, we directly assign a reference from the client code to a field. The client can easily mutate the contents of the list and manipulate our code as shown below.

List<Integer> numbers = new ArrayList<>();

Items items = new Items(numbers);

numbers.add(1); // This will change how items behaves as well

Instead, consider cloning the reference or creating a new reference and then assigning it to the field as shown below:

private final List<Integer> items;

public Items(List<Integer> items) {

this.items = new ArrayList<>(items);

}

The same rule applies while returning references. You need to be cautious so as not to expose internal mutable state.

14. **Handle exceptions with care**

While catching exceptions, if you have multiple catch blocks, ensure that the sequence of catch blocks is most specific to least. In the snippet below, the exception will never be caught in the second block since the Exception class is the mother of all exceptions.

try {

stack.pop();

} catch (Exception exception) {

// handle exception

} catch (StackEmptyException exception) {

// handle exception

}

If the situation is recoverable and can be handled by the client (the consumer of your library or code) then it is good to use checked exceptions. e. g. IOException is a checked exception that forces the client to handle the scenario and in case the client chooses to re-throw the exception then it should be a conscious call to disregard the exception.

15.**Ponder over the choice of data structures**

Java collections provide ArrayList, LinkedList, Vector, Stack, HashSet, HashMap, Hashtable. It's important to understand the pros and cons of each to use them in the correct context. A few hints to help you make the right choice:

Map: Useful if you have unordered items in the form of key, value pairs and require efficient retrieval, insertion, and deletion operations. HashMap, Hashtable, LinkedHashMap are all implementations of the Map interface.

List: Very commonly used to create an ordered list of items. This list may contain duplicates. ArrayList is an implementation of the List interface. A list can be made thread-safe using Collections.synchronizedList thus removing the need for using Vector. Here's some more info on why Vector is essentially obsolete.

Set: Similar to list but does not allow duplicates. HashSet implements the Set interface.

16. **Think twice before you expose**

There are quite a few access modifiers to choose from in Java — public, protected, private. Unless you want to expose a method to the client code, you might want to keep everything private by default. Once you expose an API, there's no going back.

For instance, you have a class Library that has the following method to checkout a book by name:

public checkout(String bookName) {

Book book = searchByTitle(availableBooks, bookName);

availableBooks.remove(book);

checkedOutBooks.add(book);

}

private searchByTitle(List<Book> availableBooks, String bookName) {

...

}

If you do not keep the searchByTitle method private by default and it ends up being exposed, other classes could start using it and building logic on top of it that you may have wanted to be part of the Library class. It could break the encapsulation of the Library class or it may be impossible to revert/modify later without breaking someone else's code. Expose consciously!

17**. Code to interfaces**

If you have concrete implementations of certain interfaces (e. g. ArrayList or LinkedList) and if you use them directly in your code, then it can lead to high coupling. Sticking with the List interface enables you to switch over the implementation any time in the future without breaking any code.

public Bill(Printer printer) {

this.printer = printer;

}

new Bill(new ConsolePrinter());

new Bill(new HTMLPrinter());

In the above snippet, using the Printer interface allows the developer to move to another concrete class HTMLPrinter.

18. **Don't force fit interfaces**

Take a look at the following interface:

interface BookService {

List<Book> fetchBooks();

void saveBooks(List<Book> books);

void order(OrderDetails orderDetails) throws BookNotFoundException, BookUnavailableException;

}

class BookServiceImpl implements BookService {

...

Is there a benefit of creating such an interface? Is there a scope for this interface being implemented by another class? Is this interface generic enough to be implemented by another class? If the answer to all these questions is no, then I'd definitely recommend avoiding this unnecessary interface that you'll have to maintain in the future. Martin Fowler explains this really well in his blog.

Well then, what's a good use case for an interface? Let's say we have a class Rectangle and a class Circle that has behavior to calculate perimeter. If there is a requirement, to sum up, the perimeter of all shapes — a use case for polymorphism, then having the interface would make more sense, as shown below.

interface Shape {

Double perimeter();

}

class Rectangle implements Shape {

//data members and constructors

@Override

public Double perimeter() {

return 2 \* (this.length + this.breadth);

}

}

class Circle implements Shape {

//data members and constructors

@Override

public Double perimeter() {

return 2 \* Math.PI \* (this.radius);

}

}

public double totalPerimeter(List<Shape> shapes) {

return shapes.stream()

.map(Shape::perimeter)

.reduce((a, b) -> Double.sum(a, b))

.orElseGet(() -> (double) 0);

}

19. **Override hashCode when overriding equals**

Objects that are equal because of their values are called value objects. e. g. money, time. Such classes must override the equals method to return true if the values are the same. The equals method is usually used by other libraries for comparison and equality checks; hence overriding equals is necessary. Each Java object also has a hash code value that differentiates it from another object.

class Coin {

private final int value;

Coin(int value) {

this.value = value;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Coin coin = (Coin) o;

return value == coin.value;

}

}

In the above example, we have overridden only the equals method of Object.

HashMap<Coin, Integer> coinCount = new HashMap<Coin, Integer>() {{

put(new Coin(1), 5);

put(new Coin(5), 2);

}};

//update count for 1 rupee coin

coinCount.put(new Coin(1), 7);

coinCount.size(); // 3 why?

We would expect coinCount to update the number of 1 rupee coins to 7 since we override equals. But HashMap internally checks if the hash code for 2 objects is equal and only then proceeds to test equality via the equals method. Two different objects may or may not have the same hash code but two equal objects must always have the same hash code, as defined by the contract of the hashCode method. So checking for hash code first is an early exit condition. This implies that both equals and hashCode methods must be overridden to express equality.