← All posts

Insights Java Code Review

Guidelines for Java code reviews

M Meenakshi • April 1, 2021

Having another pair of eyes scan your code is always useful and helps you spot mistakes before you break production. You need not be an expert to review someone's code. Some experience with the programming language and a review checklist should help you get started. We've put together a list of things you should keep in mind when you're reviewing Java code. Read on!

1. 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!

2. Replace imperative code with lambdas and streams

If you're using Java 8+, replacing loops and extremely verbose

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());
```

3. 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;
```

```
    }
}
```

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

building the code. For instance, passing a `@Nullable` argumen
method that accepts `@NonNull` parameters.

4. 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.

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.

6. 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.

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.

7. 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

8. 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`.

9. 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, BookUna
}

class BookServiceImpl implements BookService {
...
```

Is there a benefit of creating such an interface? Is there a scope

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))
```

10. 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);
}};
```

```
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.

If you write or review Java code, DeepSource can help you with automating the code reviews and save you a ton of time. Just add a `.deepsource.toml` file in the root of the repository and DeepSource will pick it up for scanning right away. The scan will find scope for improvements across your code and help you fix them with helpful descriptions.

Sign up and see for yourself!

More from DeepSource

Python code review checklist

JavaScript best practices to improve code quality

Learn how some of the new features in JavaScript can help you write cleaner code.

Insights Javascript Code Review

Get started with DeepSource

DeepSource is free forever for small teams and open-source projects. Start analyzing your code in less than 2 minutes.

CHOOSE AN ACCOUNT

GitHub

GitLab

Bitbucket

Newsletter

Read product updates, company announcements, how we build DeepSource, what we think about good code, and more.

Your email address