Java Best practices

1. Using naming conventions

According to Robert Martin (the author of Clean Code), an identifier (a class, a method, and a variable) should have the following characteristics:

- Self-explanatory: a name must reveal its intention so everyone can easily understand and change the code. For example, the names dor str do not reveal anything; however, the names daysToExpire or inputText do reveal their intention. Note that if a name requires a comment to describe itself, then the name is not self-explanatory.
- Meaningful distinctions: If names must be different, then they should also mean something different. For example, the names a1 and a2 are meaningless distinctions; and the names source and destination are meaningful distinctions.
- Pronounceable: names should be pronounceable as naturally as spoken language because we are humans - very good at words. For example, which name can you pronounce and remember easily: genStamp or generationTimestamp?

Here are some general naming rules:

- Class and interface names should be nouns, starting with an uppercase letter.
- Variable names should be nouns, starting with a lowercase letter.
- Method names should be verbs, starting with a lowercase letter.
- Constant names should have all UPPERCASE letters and words separated by underscores. For example: `MAX_SIZE`, `MIN_WIDTH`, `MIN_HEIGHT`, etc.
- Using `camelCase` notation for names

2. Ordering class members by the scope

We should sort the members by the visibility of the access modifiers: private, default (package), protected, and public. And each group is separated by a blank line.

```
public class StudentManager {
    private String errorMessage;
    private int numberOfColumns;
    private int numberOfRows;

    float columnWidth;
    float rowHeight;

    protected String[] columnNames;
    protected List<Student> listStudents;

    public int numberOfStudents;
    public String title;
}
```

3. Class members should be private

According to Joshua Bloch (author of Effective Java), we should minimize the accessibility of class members (fields) as inaccessible as possible.

4. Using underscores in numeric letters

```
        int maxUploadSize = 20_971_520;
        long accountBalance = 1_000_000_000_000L;
        float pi = 3.141_592_653_589F;
```

5. Avoid empty catch blocks

Generally, we should do the following things when catching an exception:

- Inform the user about the exception, e.g., tell them to re-enter inputs or show an error message. This is strongly recommended.
- Log the exception using JDK Logging or Log4J.
- Wrap and re-throw the exception under a new exception.

6. Use StringBuilder or StringBuffer instead of String concatenation

The following code snippet uses the + operator to build a SQL query:

```
String sql = "Insert Into Users (name, email, pass, address)";
sql += " values ('" + user.getName();
sql += "', '" + user.getEmail();
sql += "', '" + user.getPass();
sql += "', '" + user.getAddress();
sql += "')";
```

With `StringBuilder`, we can re-write the above code like this

```
tringBuilder sbSql
    = new StringBuilder("Insert Into Users (name, email, pass, address)");

sbSql.append(" values ('").append(user.getName());
sbSql.append("', '").append(user.getEmail());
sbSql.append("', '").append(user.getPass());
sbSql.append("', '").append(user.getAddress());
sbSql.append("')");

String sql = sbSql.toString();
```

7. Using Enums or Constants class instead of Constant interface

It’s a very bad idea to create an interface that is solely for declaring some constants without any methods. Here’s such an interface:

```
public interface Color {
    public static final int RED = 0xff0000;
    public static final int BLACK = 0x000000;
    public static final int WHITE = 0xffffff;
}
```

It’s because the purpose of interfaces is for inheritance and polymorphism, not for static stuff like that. So, the best practice recommends us to use an enum instead.

8. Avoid redundant initialization

It’s very unnecessary to initialize member variables to the following values: `0`, `false` and `null`. Because these values are the default initialization values of member variables in Java.

9. Using interface reference to collections

When declaring collection objects, references to the objects should be as generic as possible. This is to maximize flexibility and protect the code from possible changes in the underlying collection implementations class. That means we should declare collection objects using their interfaces List, Set, Map, Queue, and Deque.

For example, the following class shows a bad usage of collection references:

```
public class CollectionsRef {

    private HashSet<Integer> numbers;

    public ArrayList<String> getList() {

        return new ArrayList<String>();
    }

    public void setNumbers(HashSet<Integer> numbers) {
        this.numbers = numbers;
    }
}
```

Look at the reference types which are collection implementation classes - this locks the code to work with only these classes HashSet and ArrayList. What if we want the method getList() to return a LinkedList and the method setNumbers() to accept a TreeSet?

The above class can be improved by replacing the class references with interface references like this:

```
public class CollectionsRef {

    private Set<Integer> numbers;

    public List<String> getList() {
        // can return any kind of List
        return new ArrayList<String>();
    }

    public void setNumbers(Set<Integer> numbers) {
        // can accept any kind of Set
        this.numbers = numbers;
    }
}
```

10. Avoid using for loop with indexes when it can be replaced with enhanced for loop