Sunil Yadav

Java Coding Conventions

Using Naming Conventions

First and foremost, before writing any code you should specify a set of naming conventions for your Java project, such as how to name classes and interfaces, how to name methods, how to name variables, how to name constants, etc. These conventions must be obeyed by all programmers in your team.

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 understand and change the code easily. For example, the names `d`or `str` do not reveal anything; however the names `daysToExpire` or `inputText` do reveal their intention clearly. 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 distinction; and the names `source` and `destination` are meaningful distinction.

- **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. For example: `Student`, `Car`, `Rectangle`, `Painter`, etc.

- Variable names should be nouns, starting with a lowercase letter. For example: `number`, `counter`, `birthday`, `gender`, etc.

- Method names should be verbs, starting with a lowercase letter. For example: `run`, `start`, `stop`, `execute`, etc.

- Constant names should have all UPPERCASE letters and words are separated by underscores. For example: `MAX_SIZE`, `MIN_WIDTH`, `MIN_HEIGHT`, etc.

- Using `camelCase` notation for names. For example: `StudentManager`, `CarController`, `numberOfStudents`, `runAnalysis`, etc.

A good reference for naming conventions is an Oracle's publication: Java Code Conventions. You can consult this document to build your own naming conventions.

Having said that, naming is very important in programming as we name everything from classes to interfaces to methods to variable to constants, etc. So do not write code just to satisfy the compiler, write code so that it is readable and can be understood by humans - first is for yourself, then for your teammates, and for other guys who end up maintaining your project.

Ordering Class Members by Scopes

The best practice to organize member variables of a class by their scopes from most restrictive to least restrictive. That means we should sort the members by the visibility of the access modifiers: private, default (package), protected, and public. And each group separated by a blank line.

For example, the following members declaration looks quite messy:

```
public class StudentManager {
    protected List<Student> listStudents;
    public int numberOfStudents;
    private String errorMessage;
    float rowHeight;
    float columnWidth;
    protected String[] columnNames;
    private int numberOfRows;
    private int numberOfColumns;
    public String title;

}
```

According to this best practice, the member declaration above should be sorted out like this:

```
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;

}
```

And the members in each group are sorted by alphabetic order. This private-first and public-last style helps us quickly locate member variables when the list grows up over times.

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. That means we should use the lowest possible access modifier (hence the `private` modifier) to protect the fields. This practice is recommended in order to enforce information hiding or encapsulation in software design.

Consider the following class whose fields are made public:

```
public class Student {
    public String name;
    public int age;
}
```

The problem with this poor design is that anyone can change the values of the fields inappropriately. For example:

```
Student student = new Student();
student.name = "";
student.age = 2005;
```

Of course we don't want the name to be empty and the age to be unrealistic. So this practice encourages us to hide the fields and allow the outside code to change them the through setter methods (or mutators). Here's an example of a better design:

```
public class Student {

    private String name;
    private int age;

    public void setName(String name) {
        if (name == null || name.equals("")) {
            throw new IllegalArgumentException("Name is
invalid");
        }

        this.name = name;
    }

    public void setAge(int age) {
        if (age < 1 || age > 100) {
            throw new IllegalArgumentException("Age is
invalid");
        }

        this.age = age;
    }
}
```

As you can see, the fields `name` and `age` are declared to be private so the outside code cannot change them directly (information hiding). And we provide two setter methods `setName()` and `setAge()` which always check for valid arguments before actually updating the fields. This ensures the fields always get appropriate values.

Avoid Empty Catch Blocks

It's a very bad habit to leave catch blocks empty, as when the exception is caught by the empty catch block, the program fails in silence, which makes debugging harder. Consider the following program which calculates sum of two numbers from command-line arguments:

```
public class Sum {
    public static void main(String[] args) {
        int a = 0;
        int b = 0;

        try {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);

        } catch (NumberFormatException ex) {
        }

        int sum = a + b;

        System.out.println("Sum = " + sum);
    }
}
```

Note that the catch block is empty. If we run this program by the following command line:

```
java Sum 123 456y
```

It will fail silently:

```
Sum = 123
```

It's because the second argument `456y` causes a `NumberFormatException` to be thrown, but there's no handling code in the catch block so the program continues with incorrect result.

Therefore, the best practice is to 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.

Depending on the nature of the program, the code for handling exception may vary. But the rule of thumb is never "eat" an exception by an empty catch block.

Here's a better version of the program above:

```
public class Sum {
    public static void main(String[] args) {
        int a = 0;
        int b = 0;

        try {
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);

        } catch (NumberFormatException ex) {
            System.out.println("One of the arguments are not number." +
                               "Program exits.");
            return;
        }

        int sum = a + b;

        System.out.println("Sum = " + sum);
    }
}
```

Using Underscores in Numeric Literals

This little update from Java 7 helps us write lengthy numeric literals much more readable. Consider the following declaration:

```
int maxUploadSize = 20971520;
long accountBalance = 1000000000000L;
float pi = 3.141592653589F;
```

And compare with this one:

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

Which is more readable?

So remember to use underscores in numeric literals to improve readability of your code.

Using StringBuilder or StringBuffer instead of String Concatenation

In Java, we use the + operator to join Strings together like this:

```
public String createTitle(int gender, String name) {
    String title = "Dear ";

    if (gender == 0) {
        title += "Mr";
    } else {
        title += "Mrs";
    }

    return title;
}
```

This is perfectly fine since only few `String` objects involved.

However, with code that involves in concatenating many Strings such as building a complex SQL statements or generating lengthy HTML text, the + operator becomes inefficient as the Java compiler creates many intermediate `String` objects during the concatenation process.

Therefore, the best practice recommends using `StringBuilder` or `StringBuffer` to replace the + operator for concatenating many `String` objects together as they modify a `String` without creating intermediate `String` objects. `StringBuilder` is a non-thread safe and `StringBuffer` is a thread-safe version.

For example, consider the following code snippet that 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:

```
StringBuilder 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();
```

Using Enums or Constant Class instead of Constant Interface

It's a very bad idea to create an interface which 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 stuffs like that. So the best practice recommends us to use an enum instead. For example:

```
public enum Color {
    BLACK, WHITE, RED
}
```

In case the color code does matter, we can update the enum like this:

```
public enum Color {

    BLACK(0x000000),
    WHITE(0xffffff),
    RED(0xff0000);

    private int code;

    Color(int code) {
        this.code = code;
    }

    public int getCode() {
        return this.code;
    }
}
```

In a complex project, we can have a class which is dedicated to define constants for the application. For example:

```
public class AppConstants {
    public static final String TITLE = "Application
Name";

    public static final int VERSION_MAJOR = 2;
    public static final int VERSION_MINOR = 4;

    public static final int THREAD_POOL_SIZE = 10;

    public static final int MAX_DB_CONNECTIONS = 50;

    public static final String ERROR_DIALOG_TITLE
= "Error";
    public static final String WARNING_DIALOG_TITLE
= "Warning";
    public static final String INFO_DIALOG_TITLE
= "Information";
}
```

So the rule of thumb is: Do not use interfaces for constants, use enums or dedicated classes instead.

Avoid Redundant Initialization (0-false-null)

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. For example, the following initialization in declaration is unnecessary:

```
public class Person {
    private String name = null;
    private int age = 0;
    private boolean isGenius = false;
}
```

This is also redundant:

```
public class Person {
    private String name;
    private int age;
    private boolean;

    public Person() {
        String name = null;
        int age = 0;
        boolean isGenius = false;
    }
}
```

Therefore, if you know the default initialization values of member variables, you will avoid unnecessary explicit initialization. See more here: Java default Initialization of Instance Variables and Initialization Blocks.

Using Interface References to Collections

When declaring collection objects, references to the objects should be as generic as possible. This is to maximize the 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()` can return a `LinkedList` and the method `setNumbers()` can accept a `TreeSet`?

The above class can be improved by replace the class references to 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;
    }
}
```

Avoid using for loops with indexes

Don't use a `for` loop with an index (or counter) variable if you can replace it with the enhanced for loop (since Java 5) or `forEach` (since Java 8). It's because the index variable is error-prone, as we may alter it incidentally in the loop's body, or we may starts the index from 1 instead of 0.

Consider the following example that iterates over an array of Strings:

```
String[] names =
{"Alice", "Bob", "Carol", "David", "Eric", "Frank"};

for (int i = 0; i < names.length; i++) {
    doSomething(names[i]);
}
```

As you can see, the index variable `i` in this for loop can be altered incidentally which may cause unexpected result. We can avoid potential problems by using an enhanced for loop like this:

```
for (String aName : names) {
    doSomething(aName);
}
```