- Coding Standard - 3 -

By Md.Tanvir Hossain Saon

---

1. Naming Convention:

1.1 Classes and Interfaces:

- Class names should be treated as nouns,capitalized first letters in all internal words, and written in mixed cases.

- Similar to class names, interface names must also be capitalized.Avoid using acronyms and abbreviations and instead use complete words.

Example:

`Classes: class Student {}`
`class Integer {}`
`class Scanner {}`
	`Interfaces : Runnable`
	`Remote`
	`Serializable`

1.2 Methods:

- Methods should be verbs, written in mixed case with each internal word's first letter uppercase and the first letter lowercase.

Example:

```
public static void main(String [] args)  {}
           void calculateTax() {}
           string getSurname() {}
```

1.3 Variables:

- Short yet descriptive names are ideal for variables.
- Should be mnemonic, meaning that it should be made to make clear to the untrained eye what its intended usage is.
- All but temporary variables should avoid using single-character variable names.
- For integers, common names for temporary variables are i, j, k, m, and n; for characters, they are c, d, and e.
- Despite the fact that both underscore (_) and dollar sign ($) characters are acceptable, variable names shouldn't begin with them.
- Names should be in mixed case.

Example:

```
string firstName int orderNumber
int[] marks;
```

1.4 Constant:

- Words should be grouped together using underscores ("_") and all capitalized.
- Float, Long, String, and other predefined classes employ a variety of constants.

Example:

```
static final int DEFAULT_WIDTH
static final int MAX_HEIGHT
```

1.5 Packages:

- A unique package name should always begin with one of the top-level domain names, such as com, edu, gov, mil, net, or org, and should always be written in all lowercase ASCII letters.
- The package name's later parts differ based on the internal naming standards of the organization.

Example:

```
java.util.Scanner ;
java.io.*;
package com.mycompany.utilities ;
```

---

2. Comment and Documentation:

2.1 Block Comments:

- Descriptions of files, procedures, data structures, and algorithms are given in block comments.
- One can use block comments before each method and at the start of each file also applicable in other contexts, such techniques.
- The amount of indentation for block comments inside a function or method should match that of the code they explain.
- To distinguish a block comment from the rest of the code, it should come before a blank line.

Example:

```
/*

 * Here is a block comment.

 */
```

2.2 Single line Comments:

- Short comments can appear on a single line indented to the level of the code that follows
- If a comment can't be written in a single line, it should follow the block comment format

Example:

```
if (condition) {/* Handle the condition. */
...
}
```

2.3. Trailing Comments:

- Very short comments can appear on the same line as the code they describe.
- Should be shifted far enough to separate them from the statements.

Example:

```
if (a == 2) {return TRUE;            /* special case */
} else {
return isPrime(a);      /* works only for odd a */
}
```

2.4 Temporary Removing Code:

- A full line or just a portion of a line can be commented out using the // comment delimiter.

Example:

```
if (foo > 1) {// Do a double-flip.
...
}
else {
return false;          // Explain why here.
}
//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

---

3. Exception Handling:

- Use exceptions only for unexpected or exceptional conditions, not for flow control situations.
- Avoid using exceptions in place of logic checks.

Example:

```
public class SimpleTryCatch {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // This will throw an
ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
                        System.out.println("Error: Division by zero
is not allowed.");
        }  }  }
```

---

4. Import Format:

- **Static imports** (if any) should come after regular imports.
- Group imports logically:
    - **Standard Java library imports** (e.g., java.util.*)
    - **Third-party library imports** (e.g., org.apache.*)
    - **Project-specific imports** (your own classes)

Example:

```
import java.util.List;
import java.util.ArrayList;

import org.apache.commons.lang3.StringUtils;

import com.myproject.MyClass;
```

---

5. URL Format:

- The java.net.URL class can be used to represent a URL (Uniform site Locator), which is a reference to a web site in Java.

Example:

```
URL url = new
URL("https://www.example.com:8080/docs/resource.html?name=test
#section1");
```

---

6. Whitespace and indentation:

- For indentation, use four spaces. To make code blocks easier to understand, leave enough white space between them.

Example:

```
if (x > y) {
    x = y;
} else {
    y = x;
}
```

---

7. Line Length:

- To improve readability, keep line lengths to 80 characters or less. Longer lines can be divided into several lines.

Example:

```
String longLine = "This is a very long line of code that
should be broken "
+
"up into multiple lines for better readability.";
```

---

8. Braces Usage:

- Even for single-line control structures, always use braces.

Example:

```
if (x > y) {
    x = y;
}
```

---

9. File Organization:

- There should only be one top-level class or interface per source file. The class name and the file name ought to coincide.

Example:

```
Student.java should only contain the Student class.
```

---

10. Favor Composition Over Inheritance:

- Instead of relying heavily on inheritance, use composition when a class needs functionality from another class, as it provides better flexibility.

Example:

```
// Inheritance (Less flexible):
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car extends Engine {
    // Car inherits Engine functionality
}

}
```

```
// Composition (More flexible):
class Engine {
    public void start() {
        System.out.println("Engine started");
    }

class Car {
    private Engine engine;

    public Car() {
        this.engine = new Engine();
    }

    public void startCar() {
        engine.start();
    }
}
```

---

11. Proper Use of Access Modifiers:

- To ensure encapsulation, use the relevant access modifier (private, protected, or public). Make fields private unless absolutely required.

Example:

```
// Bad Practice:
public int age;

// Good Practice:
private int age;

public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}
```

12. Code Modularity:

- Break code into smaller, reusable modules (methods or classes) that do one specific thing. Each method should ideally have a single responsibility.

Example:

```
// Instead of doing everything in one method:
public void processOrder() {
    checkInventory();
    calculateTotal();
    applyDiscount();
    generateInvoice();
}

// Break down into separate methods:
public void processOrder() {
    checkInventory();
    calculateTotal();
    applyDiscount();
    generateInvoice();
}

private void checkInventory() { ... }
private void calculateTotal() { ... }
private void applyDiscount() { ... }
private void generateInvoice() { ... }
```

Reference:

1. https://www.geeksforgeeks.org/java-naming-conventions/
2. https://www.thoughtco.com/using-java-naming-conventions-2034199
3. https://www.oracle.com/java/technologies/javase/codeconventions-comments.html
4. https://docs.oracle.com/cd/E82085_01/150/funtional_artifacts_guide/or-fasg-standards.htm