

CODING STANDARD-1

by

Irtifa Haider

Computer Science and Engineering Department

Jahangirnagar University

Savar, Dhaka-1342, Bangladesh.

September 17, 2024

Contents

1	Naming Conventions	1
1.1	Variables	1
1.2	Constants	2
1.3	Functions/Methods	2
1.4	Classes	3
1.5	XML Elements	3
1.6	Packages	4
2	Layout Conventions	5
2.1	Indentation	5
2.2	Empty Lines	5
2.3	Braces	6
3	Member Order	7
4	Code Comments	9
4.1	Inline Comments	9
4.2	Method and Class Comments	10
4.3	XML Layout Comments	11
5	Error Handling	12
6	Code Reusability	14

7	References	15
7.1	Why Some Conventions Were Not Chosen	15

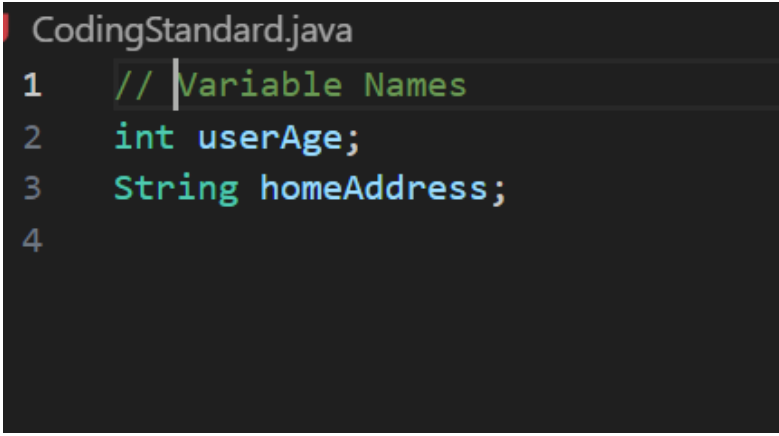
Chapter 1

Naming Conventions

1.1 Variables

Use camelCase for variable names.

Example:



```
CodingStandard.java
1  // Variable Names
2  int userAge;
3  String homeAddress;
4
```

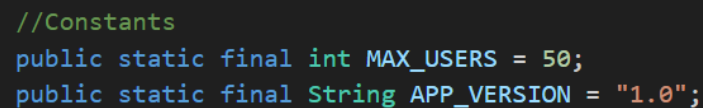
Figure 1.1

Reason: The use of capitalization in the middle of the variable helps distinguish individual words. This makes variable names easy to read and consistent with Java's common practices.

1.2 Constants

Constants should be written in `UPPERCASE_SNAKE_CASE`.

Example:



```
//Constants
public static final int MAX_USERS = 50;
public static final String APP_VERSION = "1.0";
```

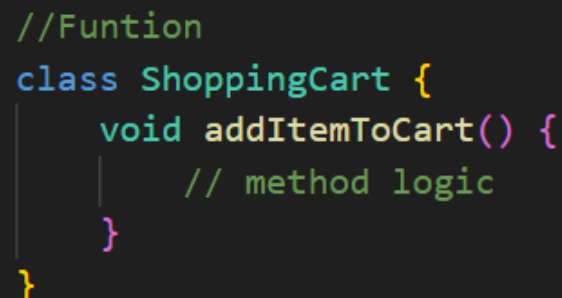
Figure 1.2

Reason: `UPPERCASE_SNAKE_CASE` makes constants stand out from other elements in the code. Since constants are meant to represent values that don't change, making them easily identifiable helps programmers quickly recognize them.

1.3 Functions/Methods

Use camelCase for method names.

Example:



```
//Funtion
class ShoppingCart {
    void addItemToCart() {
        // method logic
    }
}
```

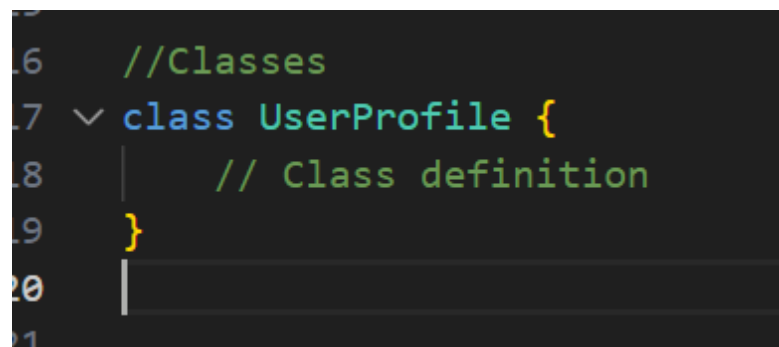
Figure 1.3

Reason: This helps in naming methods in a way that reflects their actions, keeping your code clean and understandable.

1.4 Classes

Class names should be written in PascalCase.

Example:



```
16 //Classes
17 class UserProfile {
18     // Class definition
19 }
20
21
```

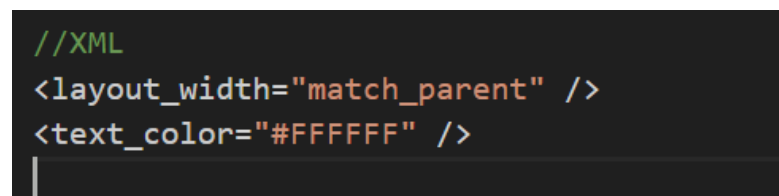
Figure 1.4

Reason: PascalCase improves readability by capitalizing the first letter of each word in the class name. This convention is also a standard practice in Java.

1.5 XML Elements

Use lowercase and separate words with underscores.

Example:



```
//XML
<layout_width="match_parent" />
<text_color="#FFFFFF" />

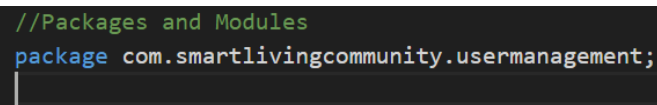
```

Figure 1.5

Reason: Using lowercase with underscores ensures that XML names remain consistent and compatible, avoiding case-sensitivity issues that might arise in different programming languages or systems, like Java. Separating words with underscores makes multi-word names more readable.

1.6 Packages

Use lowercase names, with no underscores, following the hierarchical naming structure. Example:

A screenshot of a code editor with a dark background. The first line is a comment in green: `//Packages and Modules`. The second line is a package declaration in blue: `package com.smartlivingcommunity.usermanagement;`. A vertical cursor is positioned at the end of the second line.

```
//Packages and Modules
package com.smartlivingcommunity.usermanagement;
|
```

Figure 1.6

Reason: This is the standard practice for package names in Java and Android to ensure compatibility across different platforms.

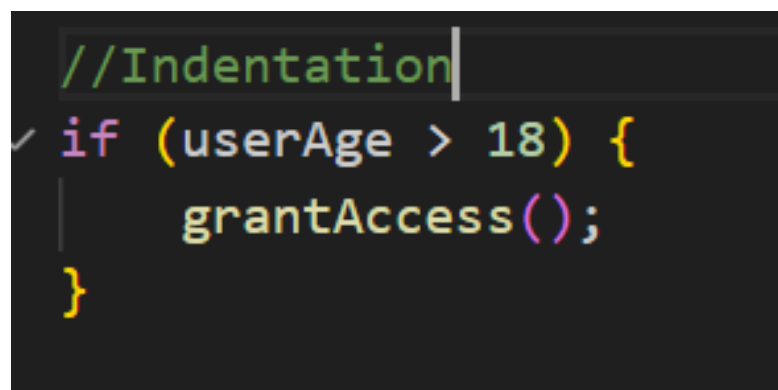
Chapter 2

Layout Conventions

2.1 Indentation

Use 4 spaces per indentation level. Do not use tabs.

Example:

A screenshot of a code editor with a dark background. The text is as follows:

```
//Indentation
if (userAge > 18) {
    grantAccess();
}
```

The code uses 4 spaces for indentation. The first line is a comment. The second line is an if-statement opening brace. The third line is the method call, indented. The fourth line is the closing brace. The text is color-coded: green for comments, purple for keywords, yellow for parentheses and semicolons, and orange for the method name.

Figure 2.1

Reason: This creates a clear visual hierarchy in code structure, making it easier to follow.

2.2 Empty Lines

Add an empty line between:

- Methods
- Blocks of unrelated code.

Example:


```
//Empty Lines
public void startService() {
    // Some code here
}

public void stopService() {
    // Other code here
}
```

Figure 2.2

Reason: Using empty lines to separate different sections of code improves readability. This also helps to separate different sections of your code.

2.3 Braces

Always put the opening brace on the same line as the statement.

Example:

```
//Braces
public void startService() {
    // Some code here
}
```

Figure 2.3

Reason: Placing the opening brace on the same line as the statement makes the code more compact and readable, reducing unnecessary vertical space.

Chapter 3

Member Order

The member order defines the structure inside a class, ensuring consistency across the codebase.

Recommended Order Inside a Class:

Constants.

Variables.

Constructors.

Methods: Public methods, followed by private methods.

Example:

```
//Member Order
public class SmartDevice {

    // 1. Constants
    public static final String DEVICE_TYPE = "Thermostat";

    // 2. Fields
    private String deviceName;
    private int deviceID;

    // 3. Constructor
    public SmartDevice(String name, int id) {
        this.deviceName = name;
        this.deviceID = id;
    }

    // 4. Methods
    public void turnOn() {
        // Code to turn on the device
    }
}
```

Figure 3.1

Reason: This ordering follows a natural top-down structure. Constants provide class-level information, fields represent the object's state, constructors initialize the state, and methods define the behavior of the class. This logical order helps in understanding and

maintaining the code structure.

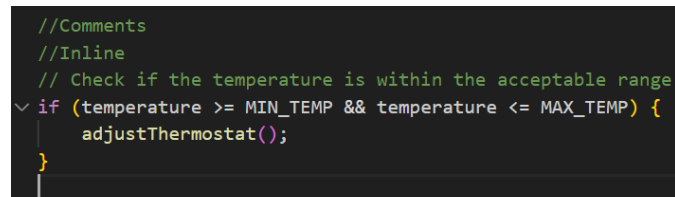
Chapter 4

Code Comments

4.1 Inline Comments

Use inline comments to clarify complex logic.

Example:

A screenshot of a code editor with a dark background. It shows a code snippet with a multi-line comment and an if statement. The comment is split across three lines: '//Comments', '//Inline', and '// Check if the temperature is within the acceptable range'. Below the comment is an if statement: 'if (temperature >= MIN_TEMP && temperature <= MAX_TEMP) {', followed by an indented line 'adjustThermostat();', and then a closing brace '}'.

```
//Comments
//Inline
// Check if the temperature is within the acceptable range
if (temperature >= MIN_TEMP && temperature <= MAX_TEMP) {
    adjustThermostat();
}
```

Figure 4.1

Reason: Using inline comments highlights why certain decisions were made.

4.2 Method and Class Comments

Each method/class should have a Javadoc comment to describe its purpose, parameters, and return value(for method).

Example:

```
//Method Comment
/**
 * Calculates the average temperature in a room.
 *
 * @param roomId the ID of the room
 * @return the average temperature in degrees Celsius
 */
public int calculateRoomTemperature(int roomId) {
    // Logic here
}

//Class Comment
/**
 * Manages all devices connected to the smart home system.
 */
public class SmartHomeManager {
    // Class implementation here
}
```

Figure 4.2

Reason: Even if the code is well-written, the purpose of a method/class and their behavior/functionality may not always be immediately clear from just the code itself. By providing a high-level overview of what the method/class does, you make it much easier for others to understand the method's functionality without needing to examine its implementation details line by line.

4.3 XML Layout Comments

Add comments in XML files to explain the section.

Example:

```
}  
//XML  
<!-- User Profile Layout -->  
<TextView  
    android:id="@+id/userName"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Name" />
```

Figure 4.3

Reason: This helps in understanding the structure and purpose of different parts of your XML configuration.

Chapter 5

Error Handling

- Use specific exceptions rather than catching or throwing generic exceptions.

Example:

```
try {
    BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
} catch (FileNotFoundException e) {
    // Handle file not found
} catch (IOException e) {
    // Handle other I/O errors
}

// Bad practice
try {
    BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
} catch (Exception e) {
    // This catches too many potential exceptions, making it unclear what went wrong
}
```

Figure 5.1

Reason: Using specific exceptions like `IOException`, `SQLException`, or `NullPointerException` provides clear information about what went wrong making error handling more precise and informative.

- Avoid empty catch blocks.

Example:

```
//2.
// Good practice
try {
    BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
} catch (FileNotFoundException e) {
    System.err.println("File not found: " + e.getMessage());
}

// Bad practice
try {
    BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
} catch (FileNotFoundException e) {
    // Nothing here, exception is silently swallowed
}
```

Figure 5.2

Reason: Avoiding empty catch blocks ensures that errors are either properly handled or logged, preventing silent failures.

Chapter 6

Code Reusability

- DRY (Don't Repeat Yourself)

Reason: Avoid duplicating code by abstracting common logic into reusable methods or classes. This keeps codebase clean and maintainable.

- Use interfaces and abstract classes.

Reason: It helps define shared behavior and promotes polymorphism. This allows you to write more flexible and reusable code.

```
// Interface
interface Animal {
    void sound(); // Abstract method for sound
}

// Abstract class
abstract class Mammal implements Animal {
    public void walk() {
        System.out.println("Walking...");
    }
}

// Dog class implementing Animal and extending Mammal
class Dog extends Mammal {
    public void sound() {
        System.out.println("Bark");
    }
}

// Main class to demonstrate polymorphism
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Polymorphism
        myDog.sound(); // Calls Dog's sound method
    }
}
```

Figure 6.1

Chapter 7

References

1. Secure Java URL Encoding and Decoding
2. Mastering Efficiency: A Guide on How to Write Reusable Java Code
3. Google Java Style Guide
4. Coding Standards and Best Practices
5. OpenAI ChatGPT
6. Java Naming Conventions
7. Using Java Naming Conventions
8. Java Code Conventions
9. Functional Artifacts Guide
10. Coding Conventions and Guidelines (PDF)

7.1 Why Some Conventions Were Not Chosen

For more information on why some conventions were not chosen, please refer to the **Why Not Us? We Are Not Standard Enough?**