

CODING STANDARD-1

by

Irtifa Haider

Computer Science and Engineering Department

Jahangirnagar University

Savar, Dhaka-1342, Bangladesh.

September 14, 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	8
4.1	Inline Comments	8
4.2	Method and Class Comments	9
4.3	XML Layout Comments	10
5	Error Handling	11
6	Code Reusability	13

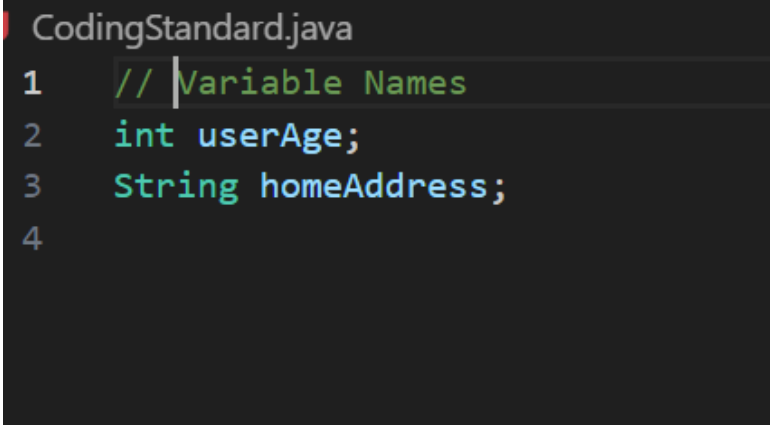
Chapter 1

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

I suggest this, because: The use of capitalization in the middle of the variable helps distinguish individual words.

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

I suggest this, because: `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
    }
}
```

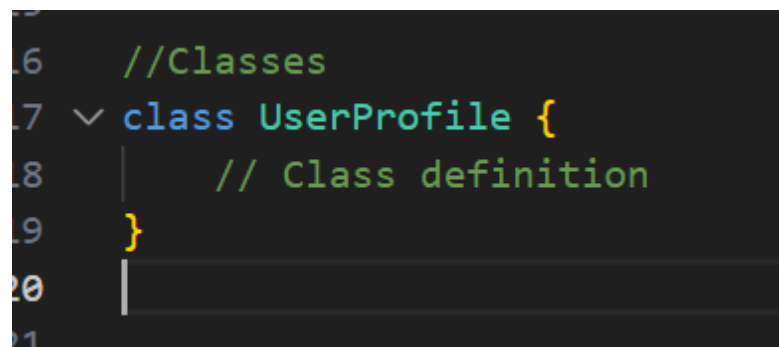
Figure 1.3

I suggest this, because: Method names typically represent actions and camelCase makes these action-oriented names easier to read.

1.4 Classes

Class names should be written in PascalCase.

Example:



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

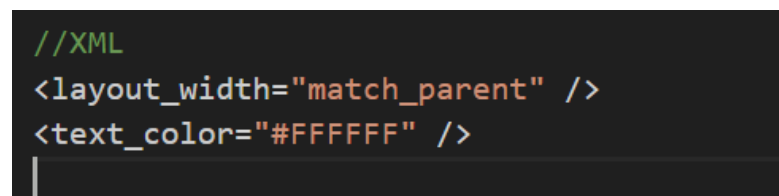
Figure 1.4

I suggest this, because: PascalCase improves readability by capitalizing the first letter of each word in the class name.

1.5 XML Elements

Use lowercase and separate words with underscores.

Example:



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

```

Figure 1.5

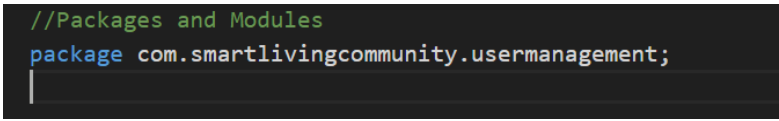
I suggest this, because: Using lowercase with underscores ensures that XML names remain consistent and compatible, avoiding case-sensitivity issues that might arise in different programming languages or systems, like: Java.

And 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 code snippet in a dark-themed editor. 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 at the end of the second line.

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

Figure 1.6

I suggest this, because: This is the standard practice for package names in Java and Android to ensure compatibility across different platforms.

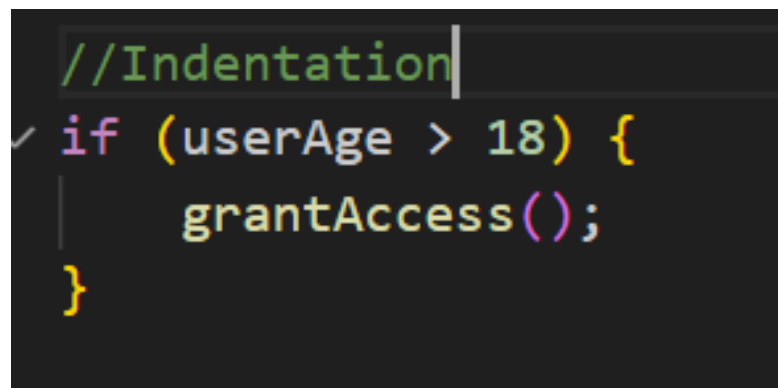
Chapter 2

Layout Conventions

2.1 Indentation

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

Example:



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

Figure 2.1

I suggest this, because: It creates a clear visual hierarchy in the code structure.

2.2 Empty Lines

Add an empty line between:

- Methods
- Blocks of unrelated code.

Example:

I suggest this, because: using empty lines to separate different sections of code for

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

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

Figure 2.2

better readability.

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

I suggest this, because: Placing the opening brace on the same line as the statement, rather than on a new line, 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

I suggest this, because: 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.

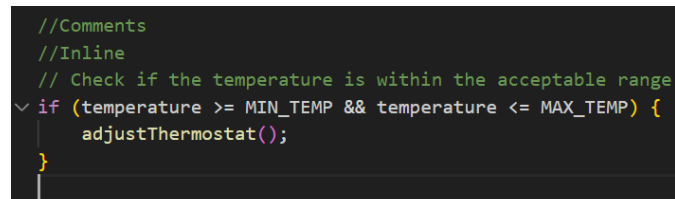
Chapter 4

Code Comments

4.1 Inline Comments

Use inline comments to clarify complex logic.

Example:



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

Figure 4.1

I suggest this, because: 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

I suggest this, because: 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

I suggest this, because: it explains why certain structures are used.

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

I suggest this, because: Using specific exceptions like `IOException`, `SQLException`, or `NullPointerException` provides clear information about what went wrong.

- 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

I suggest this, because: 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)

I suggest this, because: Avoiding duplicating code to abstract common logic into reusable methods or classes.

- Use interfaces and abstract classes.

I suggest this: to define shared behavior and promote polymorphism.

```
// 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