Coding Standard

1 Naming Convention

1.1 Classes

Use PascalCase (capitalize the first letter of each word). Example: `MainActivity`, `UserProfile`.

1.2 Methods

Use camelCase (capitalize the first letter of each word except the first one). Example: `getUserData()`, `loadSettings()`.

1.3 Variables

Use meaningful names in camelCase. Example: `userName`, `orderId`.

1.4 Constants

Use UPPER_CASE with underscores separating words. Example: `MAX_USERS`, `DEFAULT_TIMEOUT`.

1.4.1 Examples:

Classes

```
public class UserProfile {
    // Class implementation
}
```

Methods

```
public void loadSettings() {
    // Method implementation
}
```

Variables

```
String userName;
int orderId;
```

Constants

```
public static final int MAX_USERS = 100;
public static final int DEFAULT_TIMEOUT = 30;
```

2 Layout

2.1 Indentation and Spacing

- Use 4 spaces per indentation level (avoid tabs).
- Add one blank line between method declarations for readability.
- Keep line length to a maximum of 80–100 characters for readability.

2.1.1 Examples:

```
public class Example {
    private int exampleField;

    public Example() {
        this.exampleField = 0;
    }

    public void exampleMethod() {
        if (exampleField > 0) {
            // Do something
        }
    }
}
```

2.2 Braces

- Place the opening brace on the same line as the declaration (K&R style).
- Always use braces for `if`, `else`, `for`, `while`, and `do` statements, even if the block is a single line.

2.2.1 Examples:

```
public void checkValue(int value) {
    if (value > 0) {
        System.out.println("Positive value");
    } else {
        System.out.println("Non-positive value");
```

```
    }
}
```

3 Comment

3.1 Single-line comments

Use `//` for single-line comments to explain code logic.

3.2 Block comments

Use `/* */` for longer descriptions.

3.3 JavaDoc comments

Use `/** */` to document classes, methods, and functions.

3.3.1 Examples:

Single-line comments

```
// This method retrieves user data
public void getUserData() {
    // Implementation
}
```

Block comments

```
/*
 * This class handles user profiles.
 * It includes methods for loading and saving user
    data.
 */
public class UserProfile {
    // Class implementation
}
```

JavaDoc comments

```
/**
 * Retrieves the user data from the database.
 *
 * @param userId The ID of the user.
 * @return The user data.
 */
public UserData getUserData(int userId) {
    // Implementation
```

```
}
```

4 Member Order

1. Constants
2. Static Fields
3. Instance Variables
4. Constructors
5. Static Methods
6. Public Methods
7. Protected Methods
8. Private Methods
9. Getters and Setters
10. Inner Classes/Interfaces

4.0.1 Examples:

```
public class Example {
    // Constants
    public static final int MAX_USERS = 100;

    // Static Fields
    private static int instanceCount = 0;

    // Instance Variables
    private String userName;

    // Constructors
    public Example(String userName) {
        this.userName = userName;
        instanceCount++;
    }

    // Static Methods
    public static int getInstanceCount() {
        return instanceCount;
    }
```

```
    // Public Methods
    public String getUserName() {
        return userName;
    }

    // Protected Methods
    protected void setUserName(String userName) {
        this.userName = userName;
    }

    // Private Methods
    private void logUserName() {
        System.out.println("User name: " + userName);
    }

    // Getters and Setters
    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    // Inner Classes/Interfaces
    public class UserProfile {
        // Inner class implementation
    }
}
```

5 Explanation

5.1 Member Order

Constants are values that never change and apply to the whole class. Placing them at the top ensures they are easily accessible and noticeable. Static fields belong to the class rather than an instance, so keeping them near the top makes it clear which members apply across all instances. Fields represent the state of an instance of the class. Listing them before methods makes it easy to understand the structure and attributes of a class. Group them by visibility (i.e., private, protected, and public) with private fields coming first to emphasize encapsulation. Constructors initialize the object, so it's logical to place them after fields to show how an object is constructed from the fields defined above. Static methods operate on static fields or do not require an instance

of the class. Keeping them after constructors helps differentiate between class-level and instance-level functionality. Public methods define how other classes can interact with this class, so placing them next clarifies the class's external behavior. Public methods should be listed before private methods as they form the external API of the class. Protected methods may be overridden by subclasses, and listing them after public methods makes it clear which methods are intended to be used and possibly modified by child classes. Private methods are internal to the class and should be placed after public methods since they are only used by the class itself. Keeping them at the bottom keeps the focus on the public-facing API and functionality first. Some teams prefer placing getters and setters near the end or just after fields. Getters and setters are often simple methods that don't require much cognitive load, so their position is flexible. Nested classes or interfaces are placed at the end to avoid distraction from the primary functionality of the class. They're usually specialized and used within the class itself.

5.2 Naming Conventions

PascalCase and camelCase offer clear visual cues that help developers quickly identify what a name represents. PascalCase is used for classes to distinguish them as a type or object, while camelCase is used for methods and variables to show they perform an action or hold a value.

These conventions have been chosen because they break up names into readable words without needing extra symbols like underscores.

The official Java style guide and most major frameworks (like Android) endorse PascalCase for classes and camelCase for methods/variables. Sticking to these conventions helps your code integrate smoothly with libraries, APIs, and other developers' projects.