Coding Standard-02

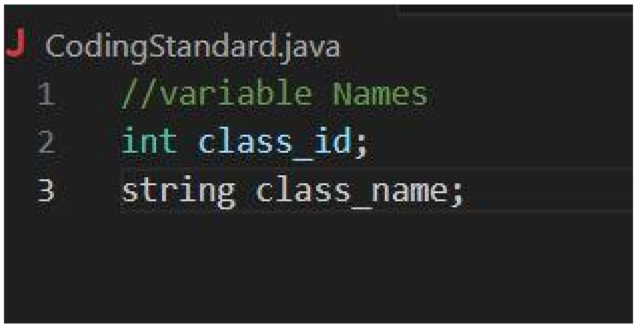
By: Mst.Solaimi Hamid

1. Naming Conventions:

1.1 Variables:

Use snake_case, It starts the name with a lowercase letter. If the name has multiple words, the later words are also lowercase, and we use an underscore (_) to separate them.

Example:



CodingStandard.java

```
//variable Names
int class_id;
string class_name;
```

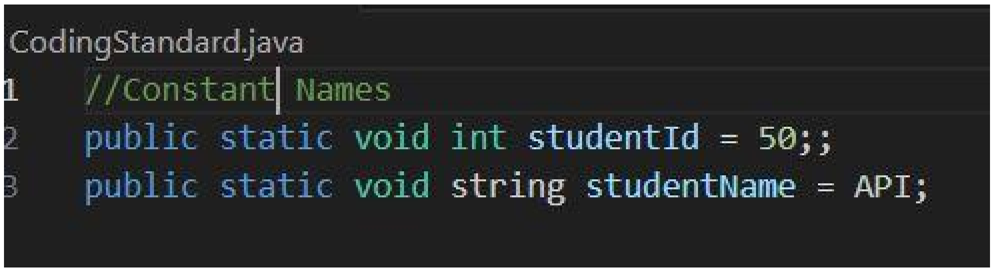
Fig 1.1

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

1.2 Constants:

Constants should be written in camelCase.

Example:



CodingStandard.java

```
//Constant Names
public static void int studentId = 50;;
public static void string studentName = API;
```

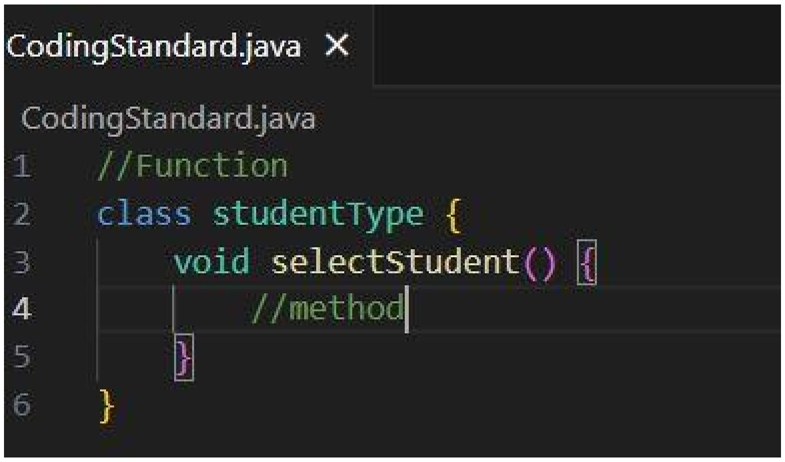
Fig 1.2

I suggest this, because: camelCase 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:



CodingStandard.java ×

CodingStandard.java

```
//Function
class studentType {
    void selectStudent() {
        //method
    }
}
```

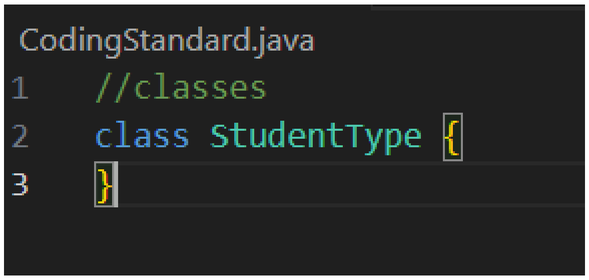
Fig 1.3

I want to use this, because Method names typically represent actions and camelCase makes these action-oriented names easier to implement.

1.4 Classes

Class names should be written in PascalCase.

Example:



CodingStandard.java

```
//classes
class StudentType {
}
```

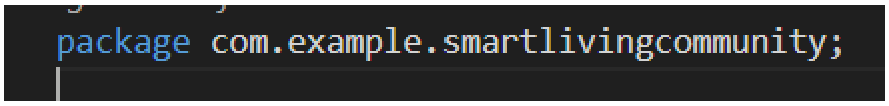
Fig 1.4

Class names and interface names must also be capitalized.Avoid using acronyms and abbreviations and instead use complete words.

1.5 Packages

Packages should be in lowercase.

Example:



```java
package com.example.smartlivingcommunity;
```

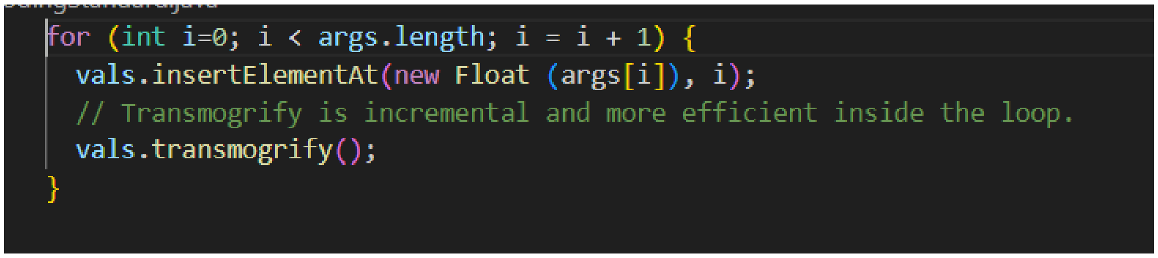
Fig 1.5

2. Layout Conventions:

2.1 Indentation

- Indent two spaces when beginning a new block.
- Open braces (i.e. "{") do not start a new line.
- Close braces (i.e. "}") do start a new line, and are indented with the code they close.
- Comments line up with the block they comment.

Example:



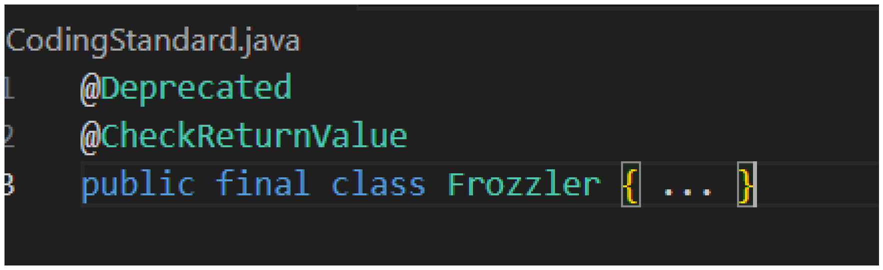
```java
for (int i=0; i < args.length; i = i + 1) {
  vals.insertElementAt(new Float (args[i]), i);
  // Transmogrify is incremental and more efficient inside the loop.
  vals.transmogrify();
}
```

Fig 2.1

2.2 Class annotations

Annotations applying to a class appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping, so the indentation level is not increased.

Example:



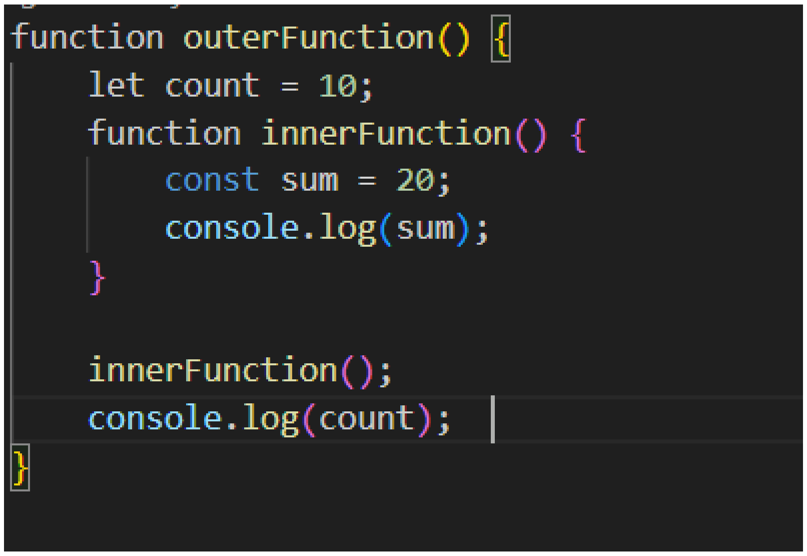
```
CodingStandard.java
@Deprecated
@CheckReturnValue
public final class Frozzler { ... }
```

Fig 2.2

2.3 Identifier

Avoid using a Single Identifier for multiple purposes.Use unique variables name.

Example:



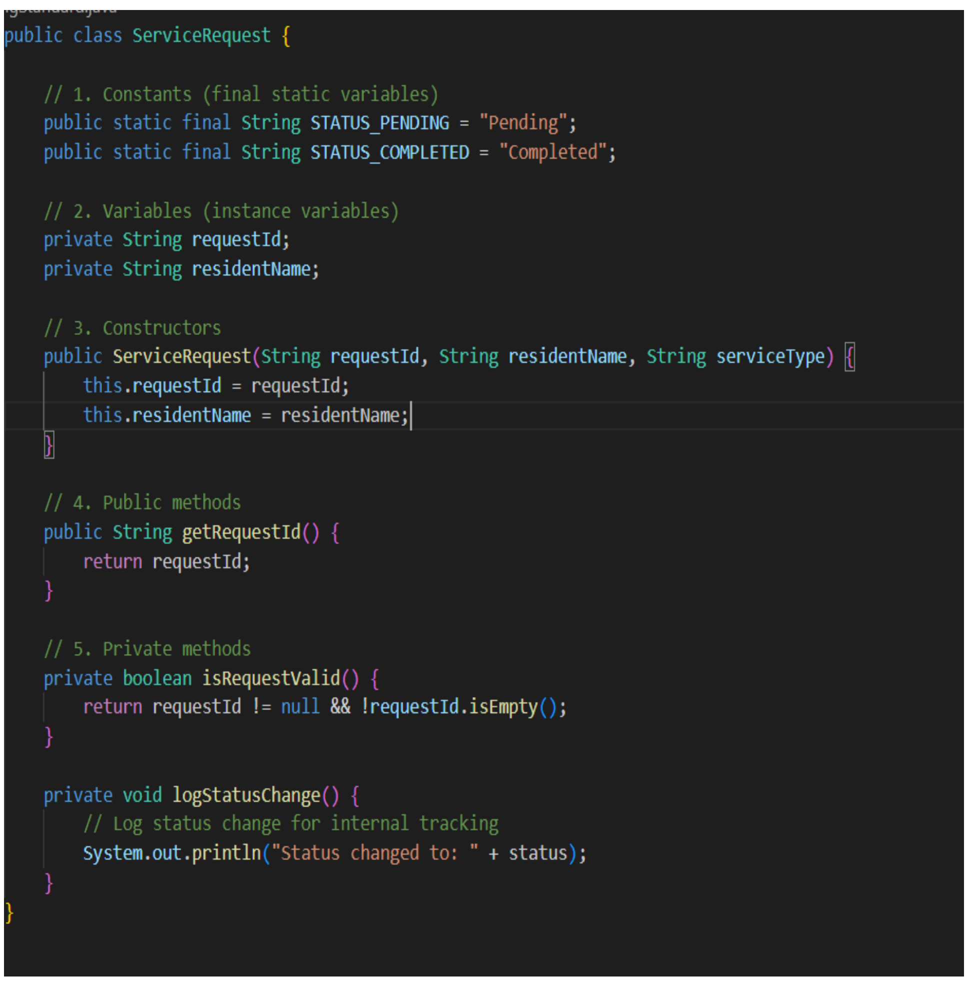
```
function outerFunction() {
    let count = 10;
    function innerFunction() {
        const sum = 20;
        console.log(sum);
    }

    innerFunction();
    console.log(count);
}
```

Fig 2.3

3. Member Order

The member order should use the following type of codebase.

Example:



```java
public class ServiceRequest {

    // 1. Constants (final static variables)
    public static final String STATUS_PENDING = "Pending";
    public static final String STATUS_COMPLETED = "Completed";

    // 2. Variables (instance variables)
    private String requestId;
    private String residentName;

    // 3. Constructors
    public ServiceRequest(String requestId, String residentName, String serviceType) {
        this.requestId = requestId;
        this.residentName = residentName;
    }

    // 4. Public methods
    public String getRequestId() {
        return requestId;
    }

    // 5. Private methods
    private boolean isRequestValid() {
        return requestId != null && !requestId.isEmpty();
    }

    private void logStatusChange() {
        // Log status change for internal tracking
        System.out.println("Status changed to: " + status);
    }
}
```

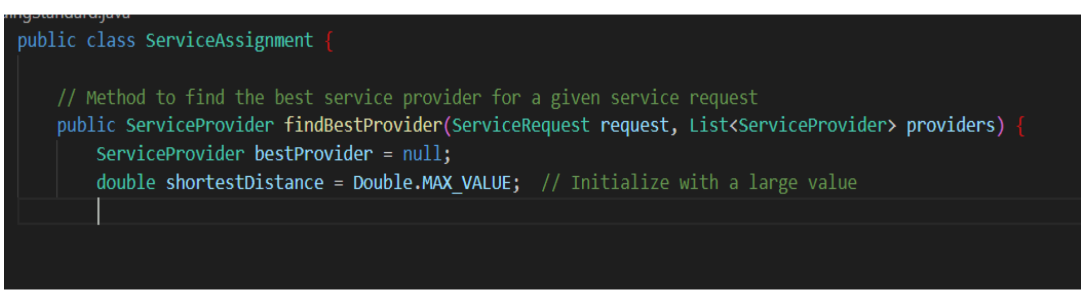
Fig 3.1

4. Code Comments

4.1 Inline Comments

Use Inline comments to clarify complex logic.

Example:



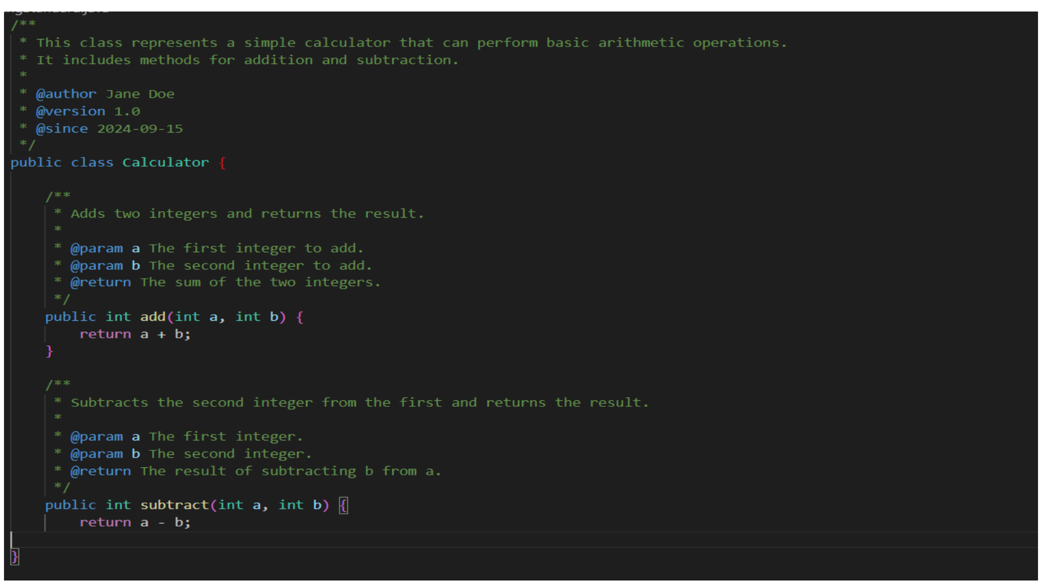
```java
public class ServiceAssignment {

    // Method to find the best service provider for a given service request
    public ServiceProvider findBestProvider(ServiceRequest request, List<ServiceProvider> providers) {
        ServiceProvider bestProvider = null;
        double shortestDistance = Double.MAX_VALUE;  // Initialize with a large value
        
```

Fig 4.1

4.2 Method and Class Comment

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

Example:



```java
/**
 * This class represents a simple calculator that can perform basic arithmetic operations.
 * It includes methods for addition and subtraction.
 *
 * @author Jane Doe
 * @version 1.0
 * @since 2024-09-15
 */
public class Calculator {

    /**
     * Adds two integers and returns the result.
     *
     * @param a The first integer to add.
     * @param b The second integer to add.
     * @return The sum of the two integers.
     */
    public int add(int a, int b) {
        return a + b;
    }

    /**
     * Subtracts the second integer from the first and returns the result.
     *
     * @param a The first integer.
     * @param b The second integer.
     * @return The result of subtracting b from a.
     */
    public int subtract(int a, int b) {
        return a - b;

}
```

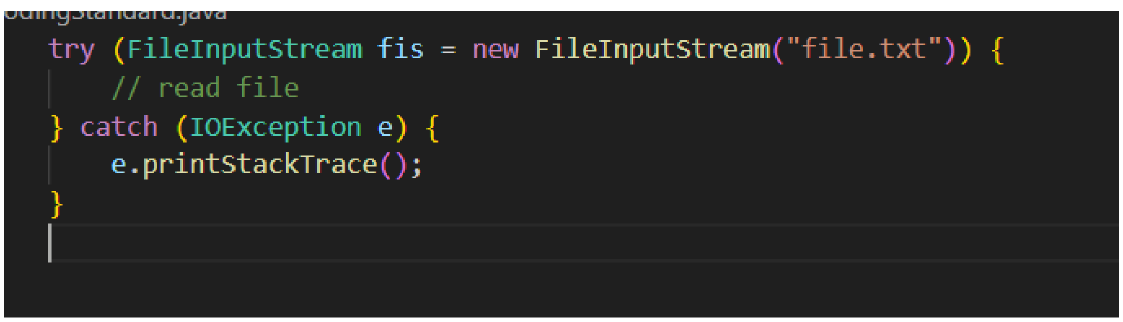
Fig 4.2

5. Exception Handling

5.1 Resource handling

Ensure that resources like files, database connections, and streams are always closed, even if an exception occurs. Use the try-with-resources statement for automatic resource management.

Example:



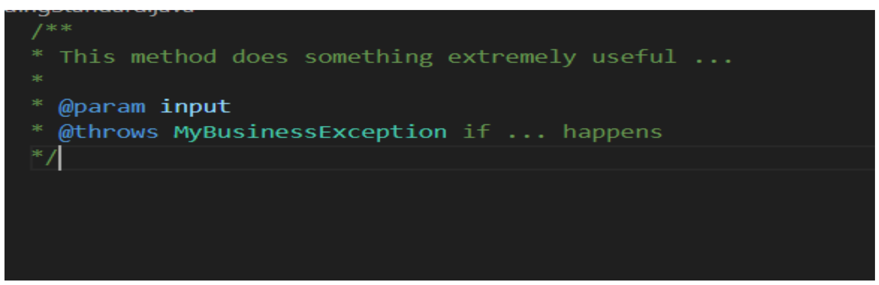
```
try (FileInputStream fis = new FileInputStream("file.txt")) {
    // read file
} catch (IOException e) {
    e.printStackTrace();
}
```

Fig 5.1

5.2 Document the Exceptions

To describe the situations that can cause the exception.

Example:



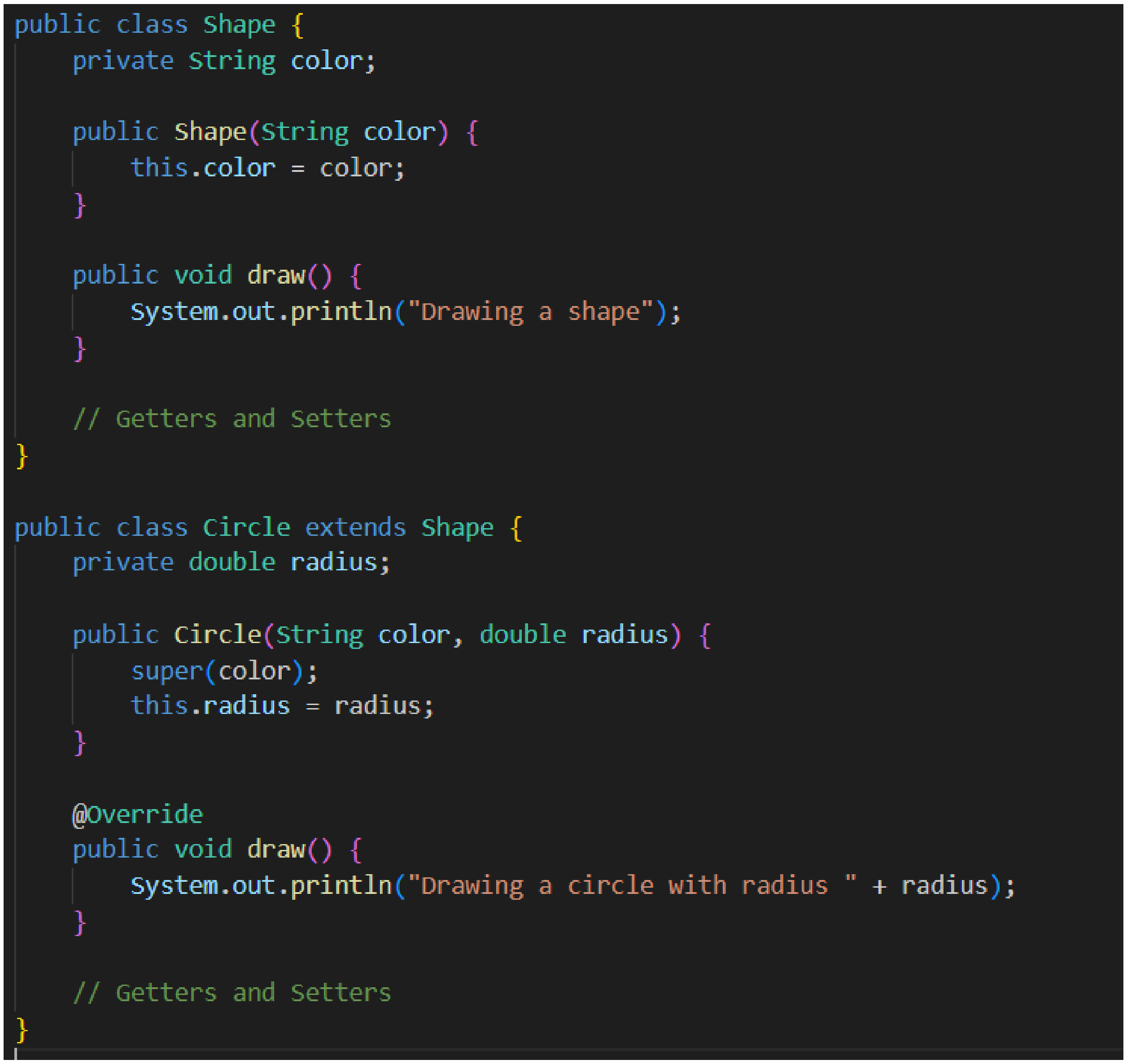
```
/**
* This method does something extremely useful ...
*
* @param input
* @throws MyBusinessException if ... happens
*/
```

Fig 5.2

6. Code Reusability

Inheritance, when used judiciously, can be a powerful tool for code reuse. Create base classes that encapsulate common functionality and derive subclasses that extend or specialize these functionalities. This hierarchical structure allows you to reuse code from parent classes while still accommodating specific requirements in derived classes.

Example:



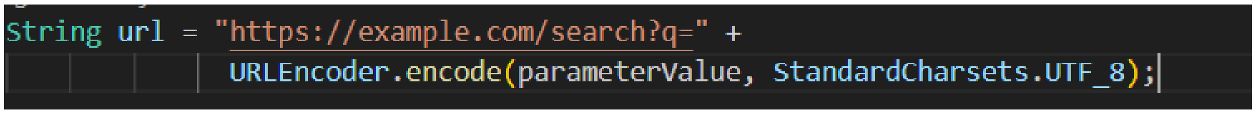
```java
public class Shape {
    private String color;

    public Shape(String color) {
        this.color = color;
    }

    public void draw() {
        System.out.println("Drawing a shape");
    }

    // Getters and Setters
}

public class Circle extends Shape {
    private double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a circle with radius " + radius);
    }

    // Getters and Setters
}
```

Fig 6.1

1. URL Encoding

encode(String s, Charset charset) has been available since Java 10 and is the best overload so far. We use a constant definition for UTF-8 (StandardCharsets.UTF_8), which eliminates the risk of typos in specifying encoding and doesn't throw any checked exceptions.

Example:



```
String url = "https://example.com/search?q=" +
        URLEncoder.encode(parameterValue, StandardCharsets.UTF_8);
```

Fig 7.1

References:
1.https://dev.to/snyk/secure-java-url-encoding-and-decoding-46o0
2.https://medium.com/@muzammilbintauseef/mastering-efficiency-a-guide-on-how-to-write-reusable-java-code-92dd2459ccc4
3.https://google.github.io/styleguide/javaguide.html#s5.2-specific-identifier-names
4.https://www.browserstack.com/guide/coding-standards-best-practices
5.https://openai.com/chatgpt/