CODEWITH_RAJESH

RAJESH KUMAR

SWIPE >>>

10 JAVA CORE BEST PRACTICES

CODEWITH_RAJESH

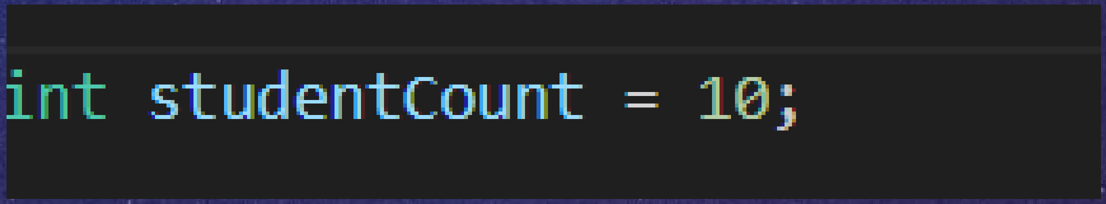
RAJESH KUMAR

SWIPE >>>

1. Use meaningful variable and method names

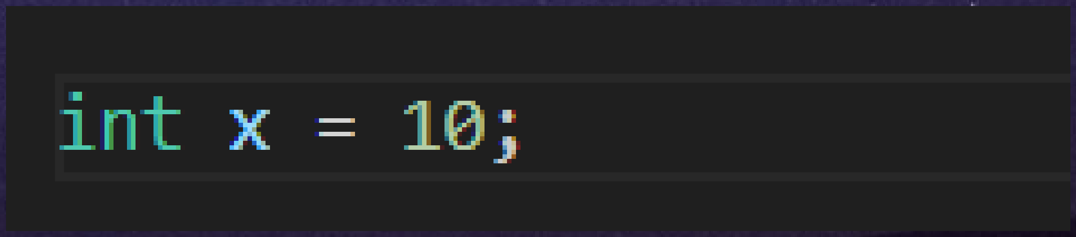
- Using descriptive names improves code readability and maintainability. In the good example, the variable **studentCount** clearly conveys its purpose, while the bad example using **x** is ambiguous and lacks clarity.

GOOD



```
int studentCount = 10;
```

BAD



```
int x = 10;
```

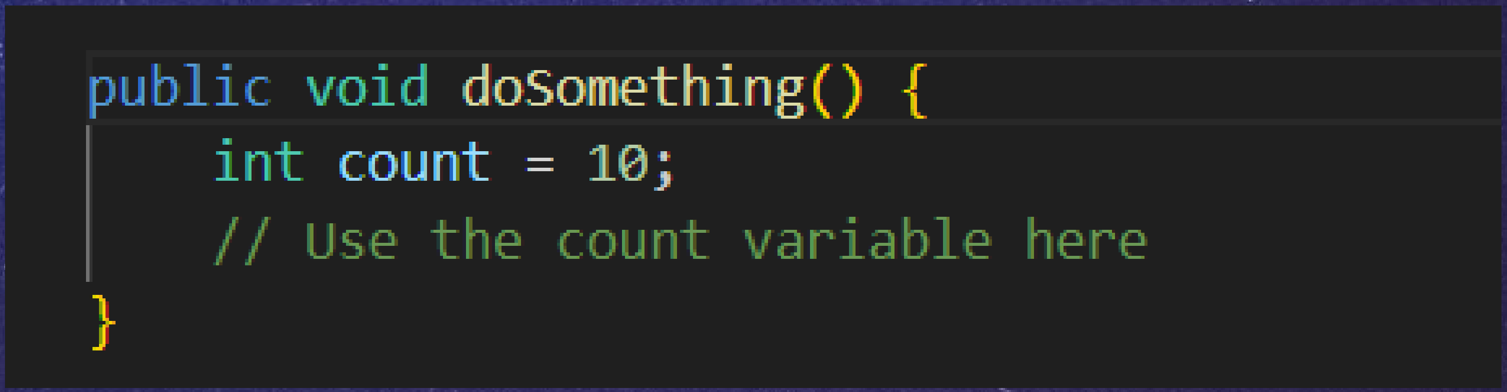
CODEWITH_RAJESH
RAJESH KUMAR

SWIPE >>>

2. Limit the scope of variables

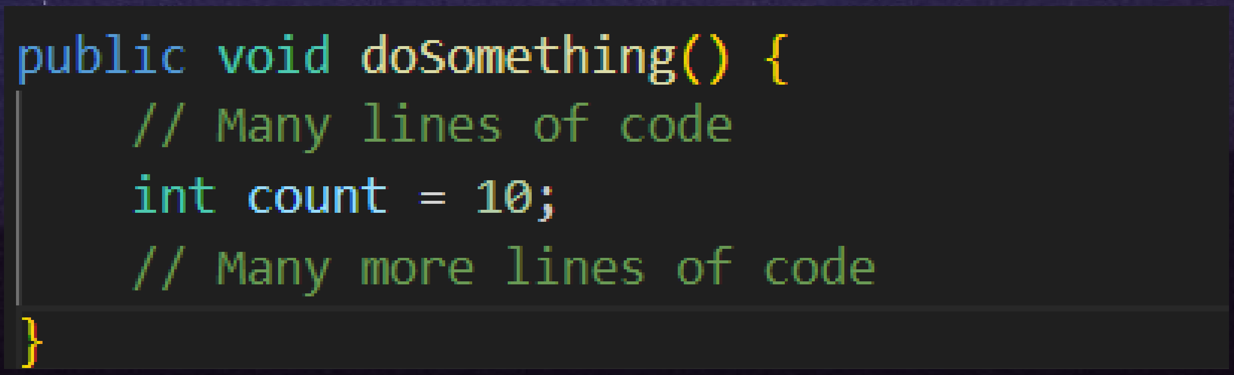
- Declaring variables closer to their usage improves code clarity and reduces the chance of accidental misuse. The good example confines the variable count within the necessary scope, while the bad example unnecessarily extends the variable's scope, making the code less maintainable.

GOOD



```java
public void doSomething() {
    int count = 10;
    // Use the count variable here
}
```

BAD



```java
public void doSomething() {
    // Many lines of code
    int count = 10;
    // Many more lines of code
}
```

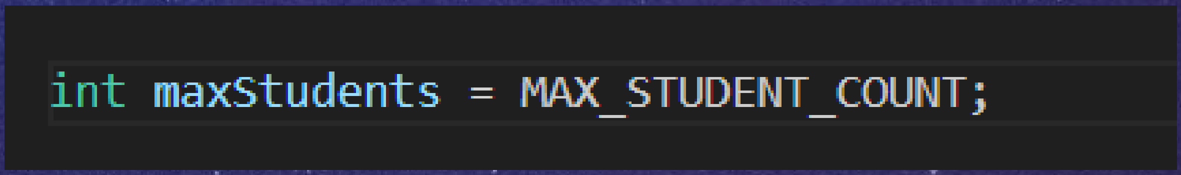
CODEWITH_RAJESH
RAJESH KUMAR

SWIPE >>>

3. Avoid magic numbers

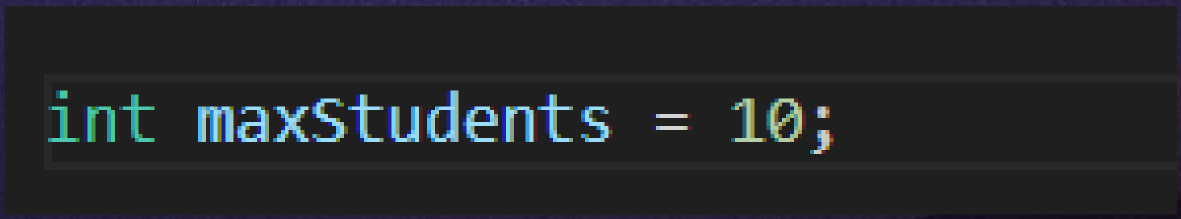
- Magic numbers are hard-coded values that lack context. Assigning them to named constants or variables enhances code readability and maintainability. The good example uses a named constant MAX_STUDENT_COUNT, while the bad example directly uses the number 10.

GOOD



```
int maxStudents = MAX_STUDENT_COUNT;
```

BAD



```
int maxStudents = 10;
```

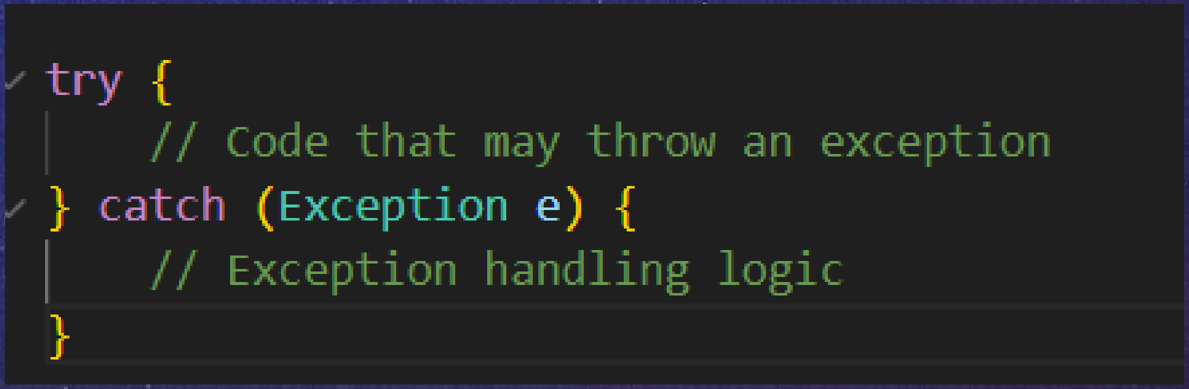
CODEWITH_RAJESH
RAJESH KUMAR

SWIPE >>>

4. Handle exceptions appropriately

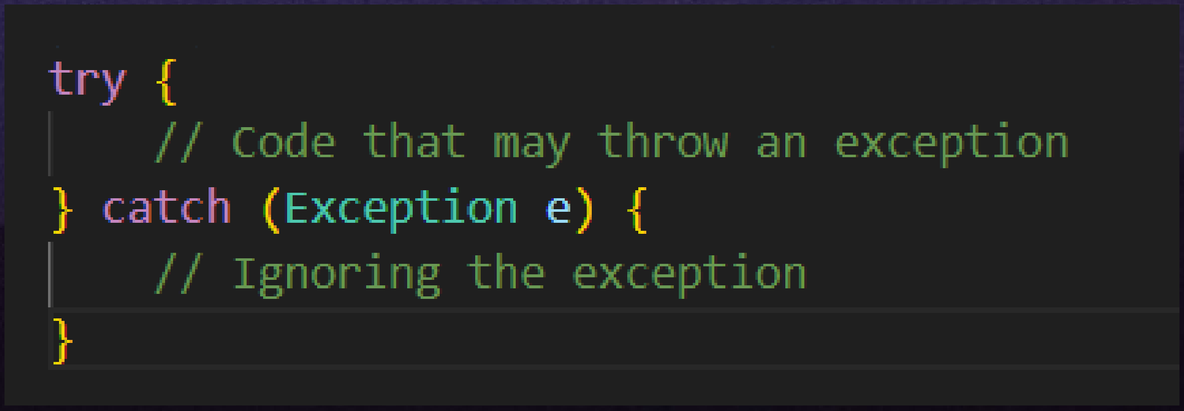
- Proper exception handling helps identify and handle errors gracefully, improving the reliability of your code. The good example includes appropriate exception handling logic, while the bad example simply ignores the exception, which can lead to hidden bugs or unexpected behavior.

GOOD



```
try {
    // Code that may throw an exception
} catch (Exception e) {
    // Exception handling logic
}
```

BAD



```
try {
    // Code that may throw an exception
} catch (Exception e) {
    // Ignoring the exception
}
```

CODEWITH_RAJESH

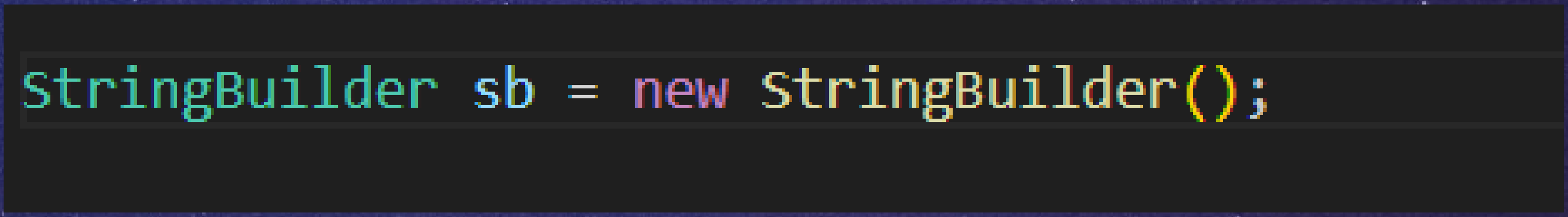
RAJESH KUMAR

SWIPE >>>

5. Avoid unnecessary object creation

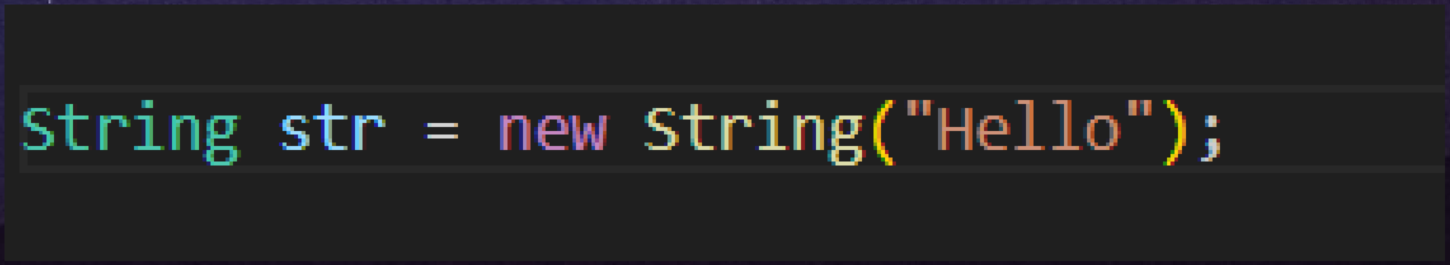
- Reusing objects, especially in loops or frequently executed code, improves performance and reduces memory overhead. The good example creates a single StringBuilder object for appending strings, while the bad example unnecessarily creates a new String object

GOOD



```
StringBuilder sb = new StringBuilder();
```

BAD

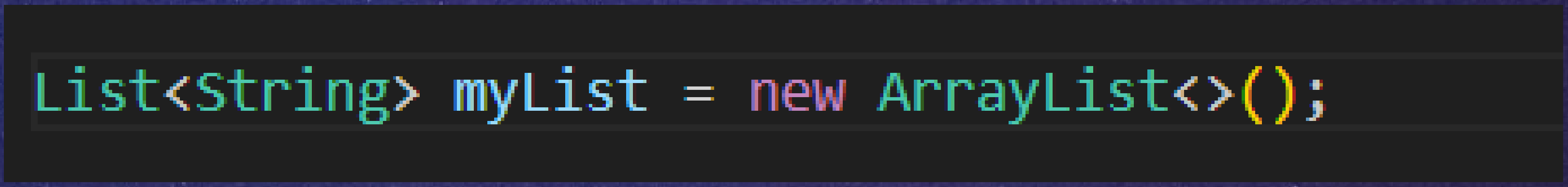


```
String str = new String("Hello");
```

6. Use interfaces and abstract classes

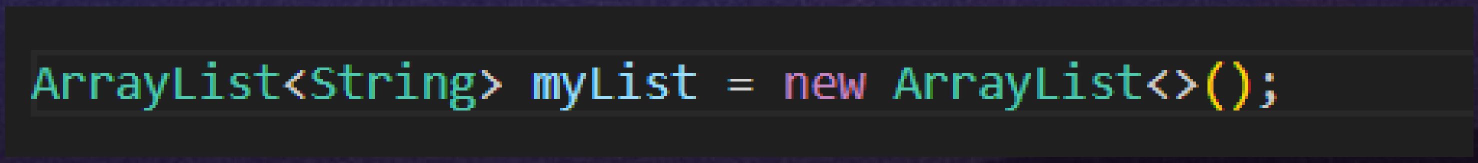
- Programming to interfaces or abstract classes instead of concrete implementations promotes flexibility and modularity. The good example uses the List interface to declare the variable myList, allowing for easier switching between different list implementations. In contrast, the bad example directly uses the concrete class ArrayList, which limits the flexibility of the code.

GOOD



```java
List<String> myList = new ArrayList<>();
```

BAD

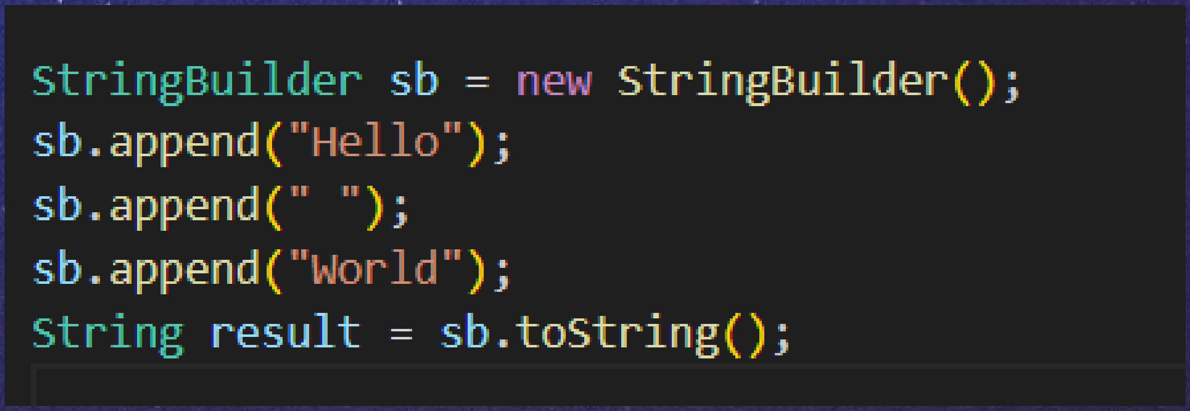


```java
ArrayList<String> myList = new ArrayList<>();
```

7. Use StringBuilder for string concatenation

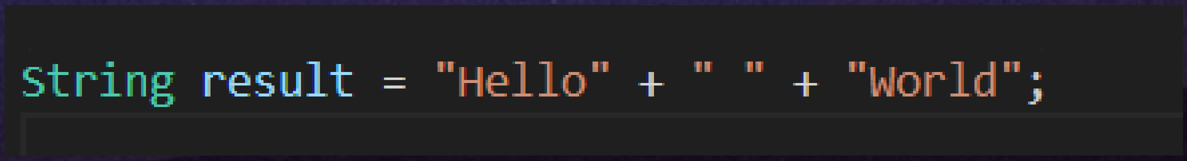
- StringBuilder is more efficient when concatenating multiple strings, especially in loops. The good example uses the StringBuilder class to efficiently concatenate multiple strings, while the bad example uses string concatenation directly with the + operator, which can lead to poor performance, particularly when concatenating multiple strings in a loop.

GOOD



```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("World");
String result = sb.toString();
```

BAD

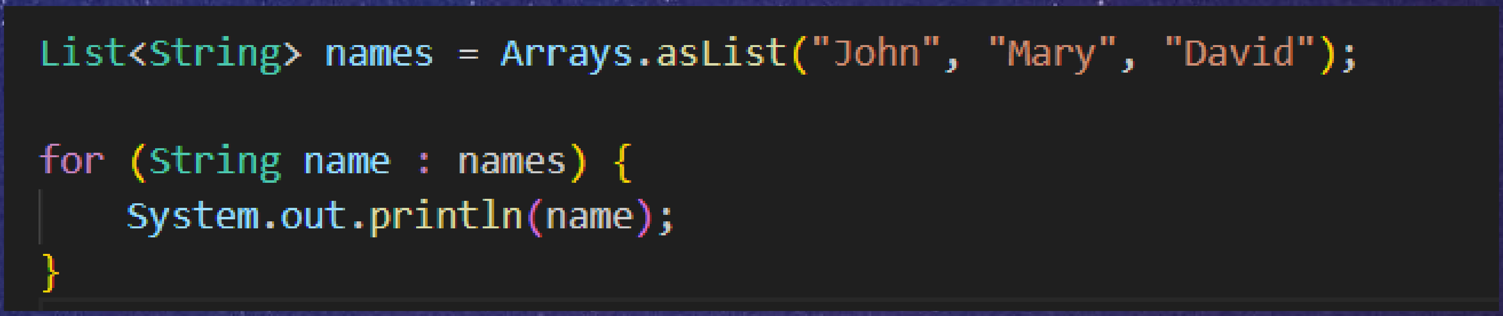


```
String result = "Hello" + " " + "World";
```

8. Good practice (enhanced for loop)

- the good practice uses enhanced for loops for simplicity and readability, while the bad practice uses for loops with indexes, which can be more error-prone and complex.

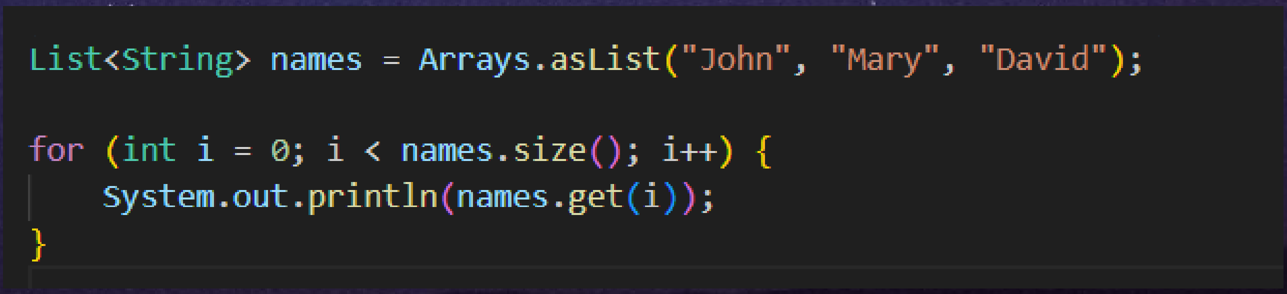
GOOD



```
List<String> names = Arrays.asList("John", "Mary", "David");

for (String name : names) {
    System.out.println(name);
}
```

BAD



```
List<String> names = Arrays.asList("John", "Mary", "David");

for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}
```

CODEWITH_RAJESH

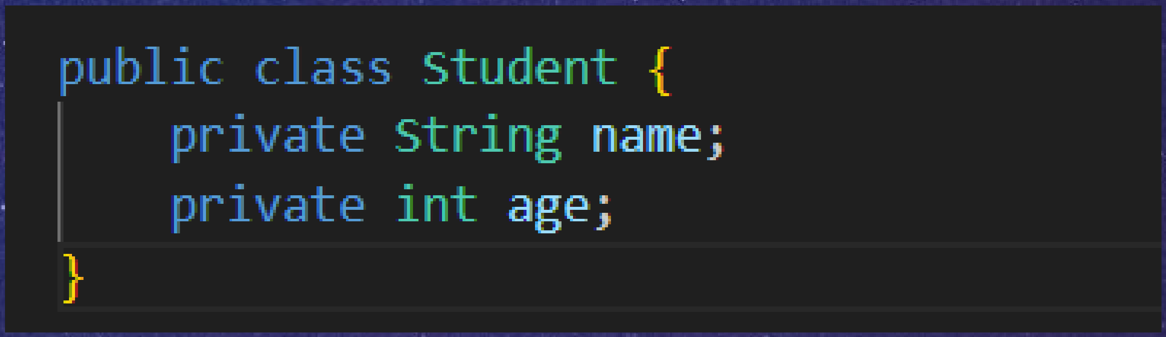
RAJESH KUMAR

SWIPE >>>

9. Good practice (private class members)

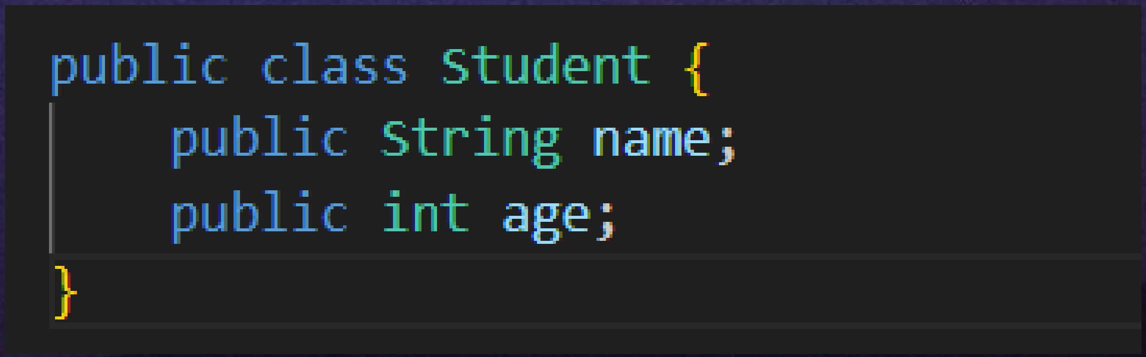
- In the good example, the name and age fields are declared as private, meaning they can only be accessed within the Student class. In the bad example, the name and age fields are declared as public, which allows direct access from anywhere in the code

GOOD



```java
public class Student {
    private String name;
    private int age;
}
```

BAD



```java
public class Student {
    public String name;
    public int age;
}
```

CODEWITH_RAJESH

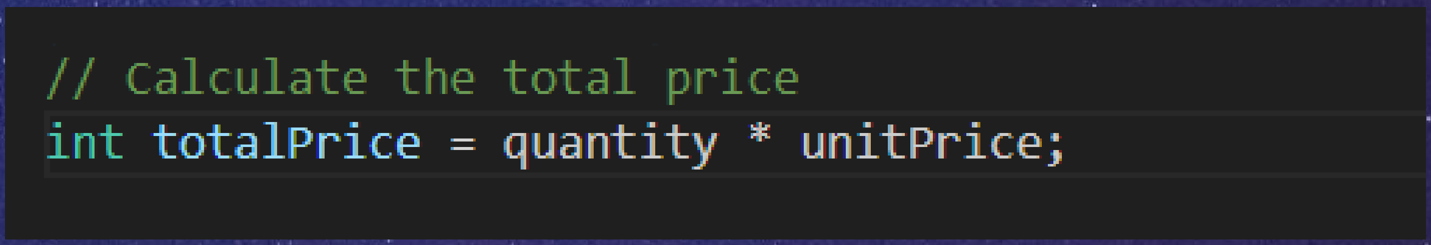
RAJESH KUMAR

SWIPE >>>

10. Use meaningful comments

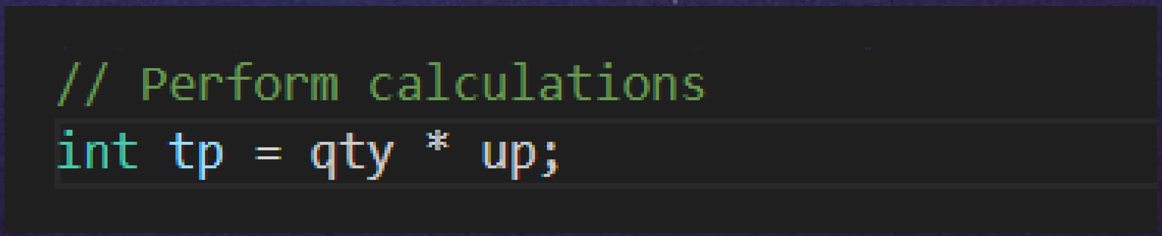
- Comments should explain the purpose or intent of the code and help other developers understand your code. The good example provides a clear comment describing the code's purpose, while the bad example has a vague comment that does not provide useful information

GOOD



```
// Calculate the total price
int totalPrice = quantity * unitPrice;
```

BAD



```
// Perform calculations
int tp = qty * up;
```

❤ Thanks for reading!

Stay up-to-date with the latest advancements in Java full stack development by following me on the handles below, where I'll be sharing my expertise and experience in the field.

codewith_rajesh

Rajesh Kumar

Like | Save | Share