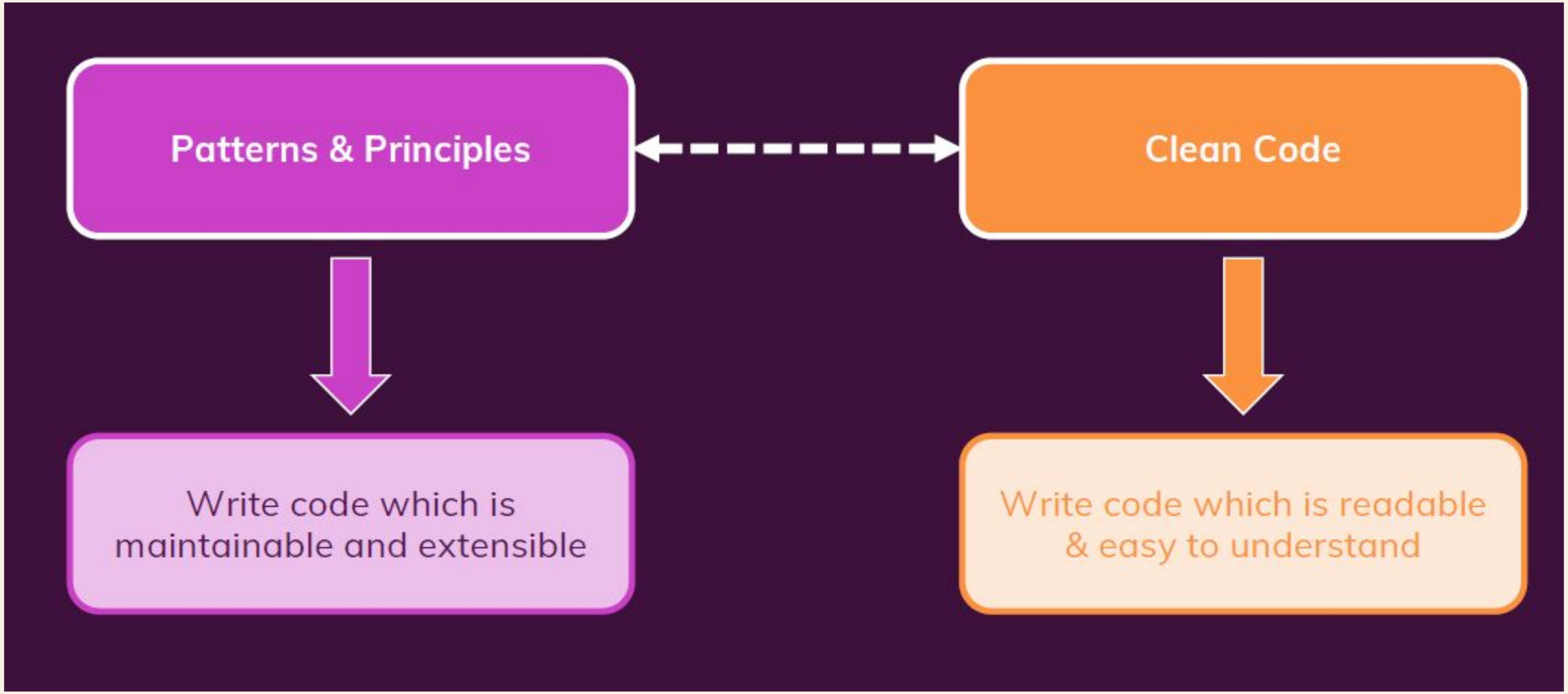
SOLID PRINCIPLES

Applying solid principles on Duolingo project

BY

VIPIN K P

Clean Code and Principle Patterns



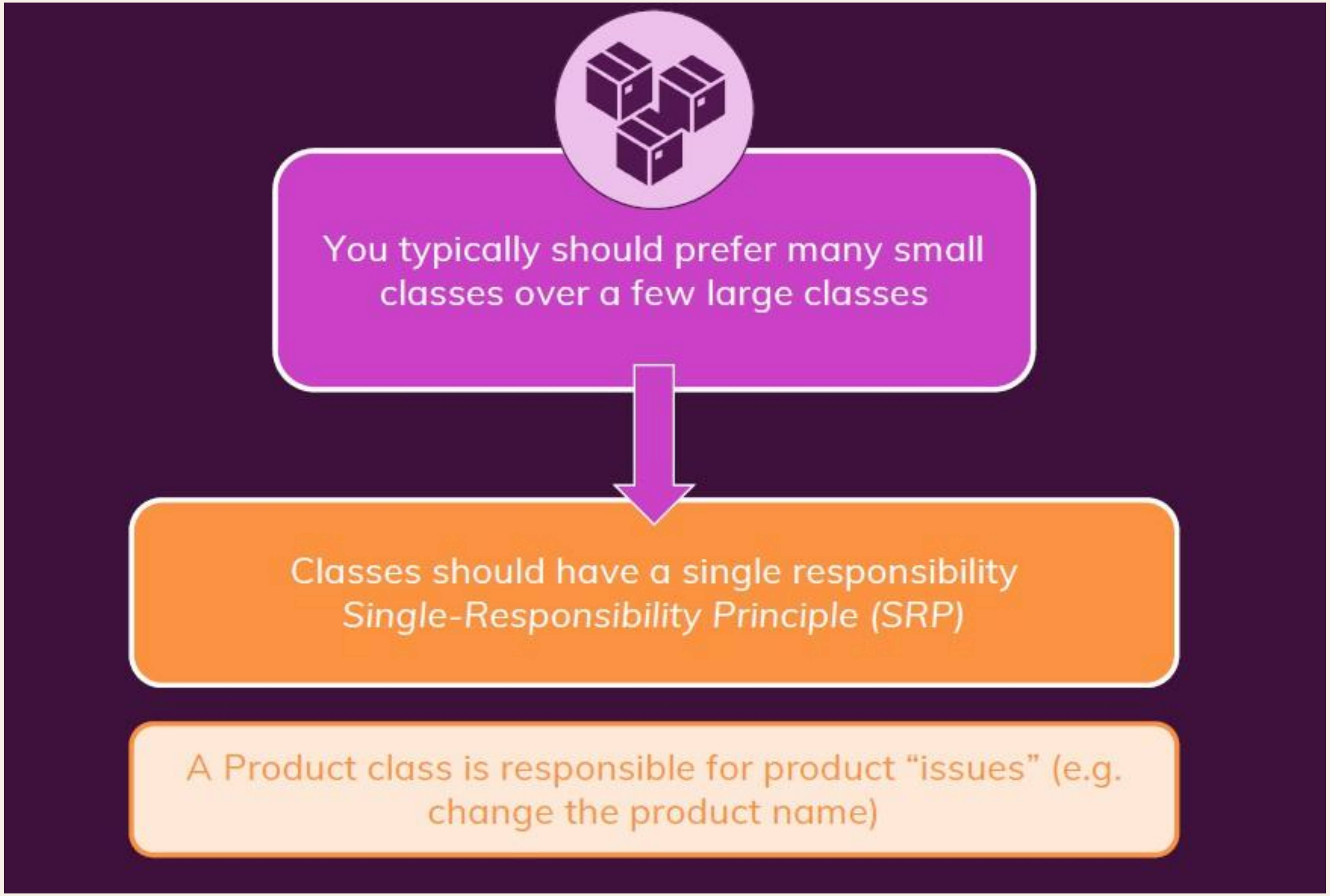
Patterns & Principles

Clean Code

Write code which is maintainable and extensible

Write code which is readable & easy to understand

Classes should be small

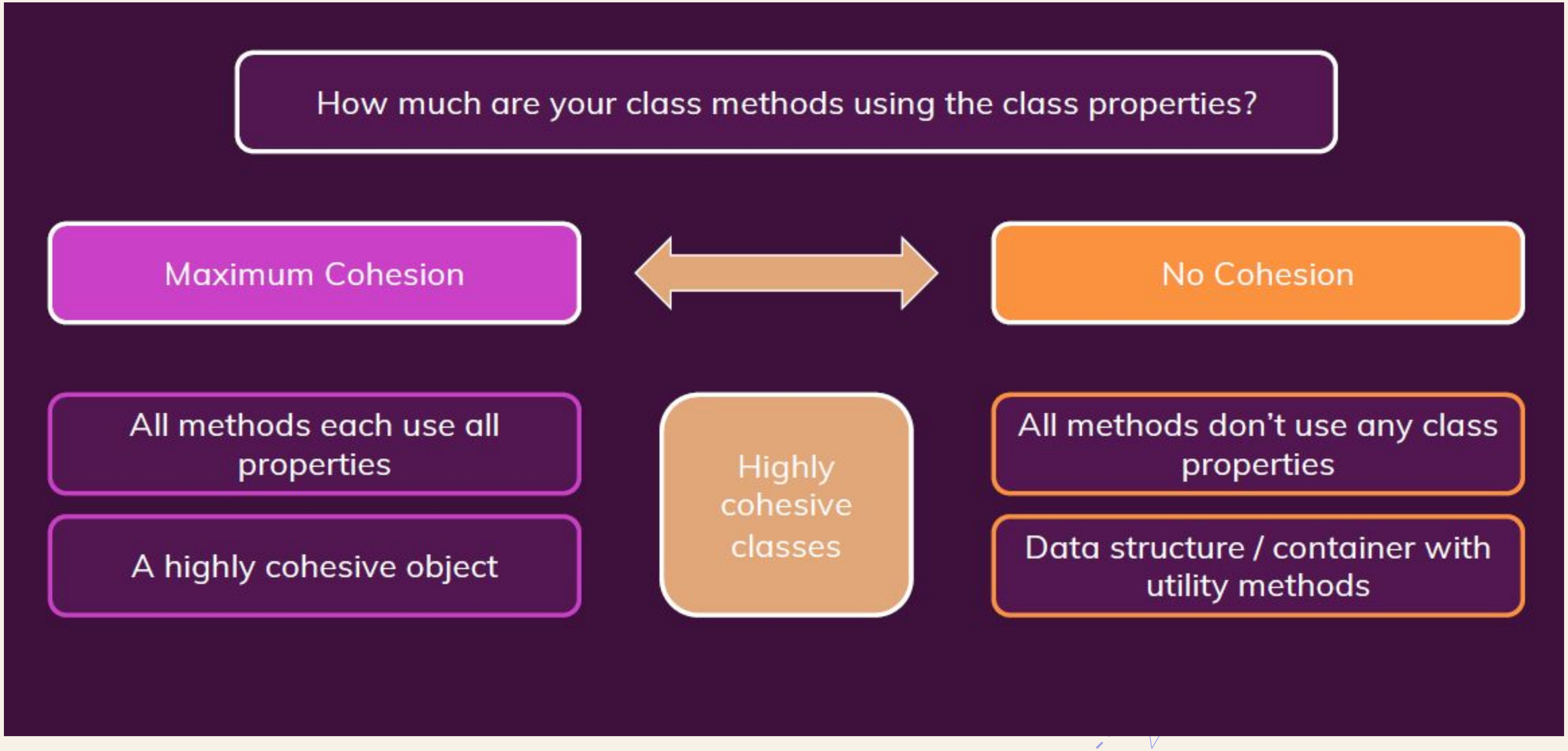


You typically should prefer many small classes over a few large classes

Classes should have a single responsibility
Single-Responsibility Principle (SRP)

A Product class is responsible for product “issues” (e.g. change the product name)

Cohesion



How much are your class methods using the class properties?

Maximum Cohesion

No Cohesion

All methods each use all properties

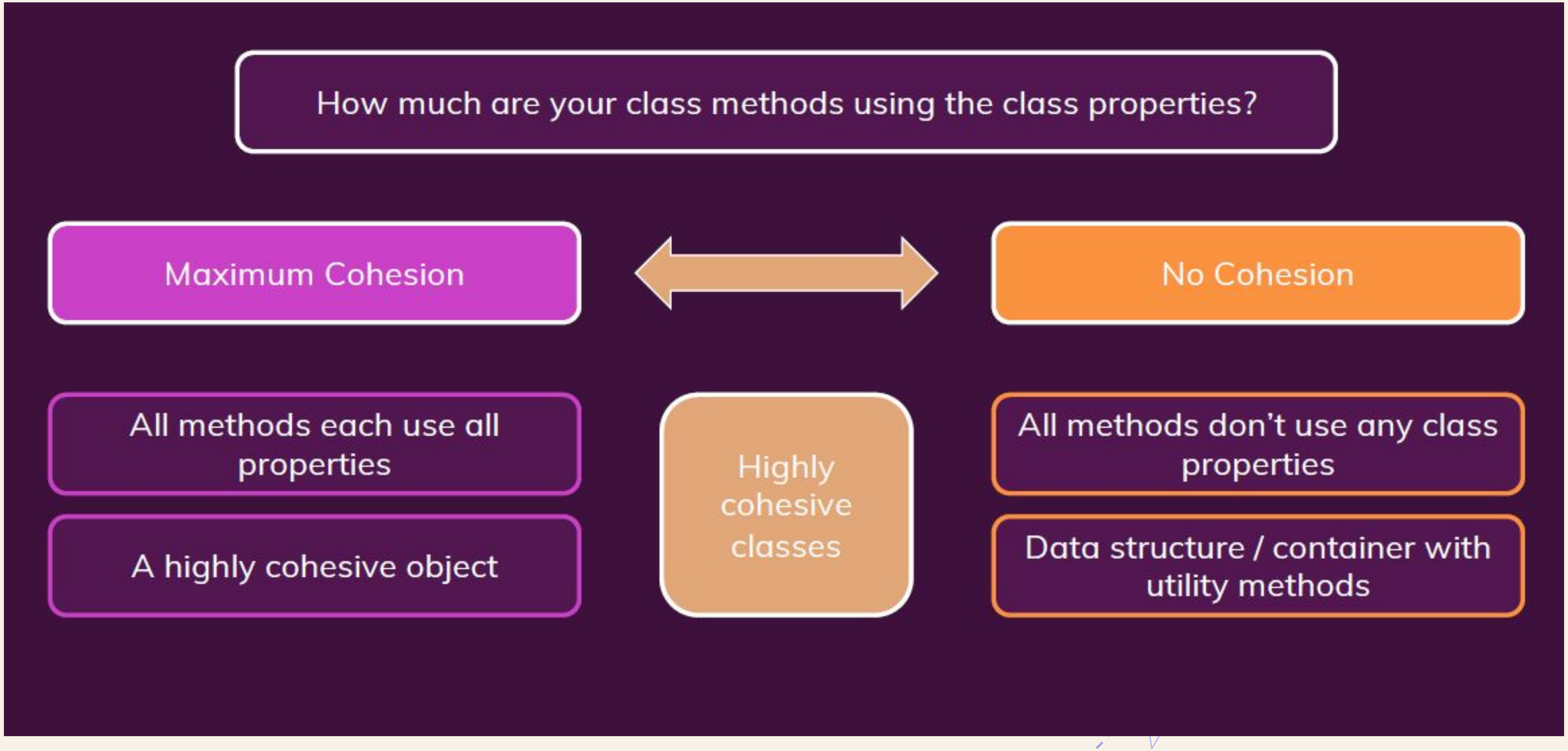
A highly cohesive object

Highly cohesive classes

All methods don't use any class properties

Data structure / container with utility methods

Cohesion



How much are your class methods using the class properties?

Maximum Cohesion

No Cohesion

All methods each use all properties

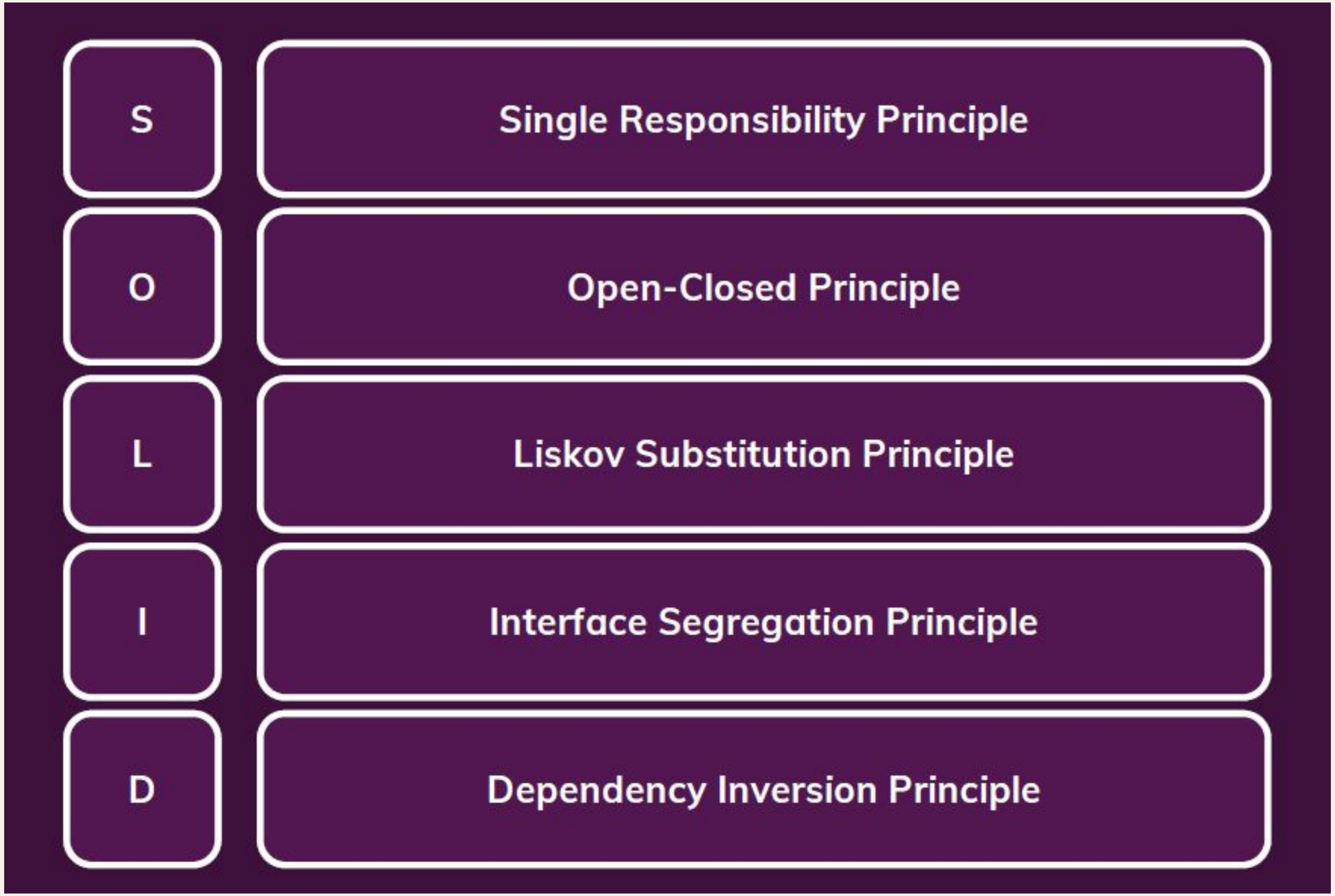
A highly cohesive object

Highly cohesive classes

All methods don’t use any class properties

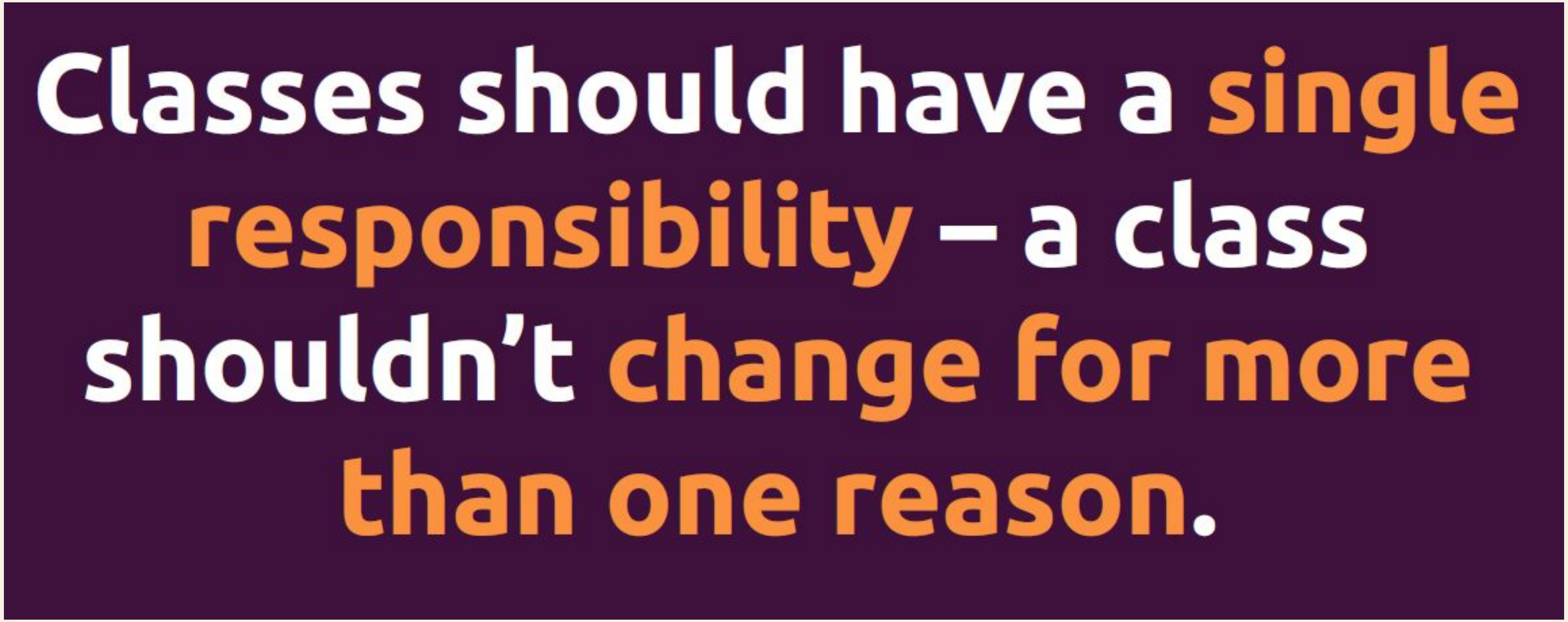
Data structure / container with utility methods

SOLID PRINCIPLES



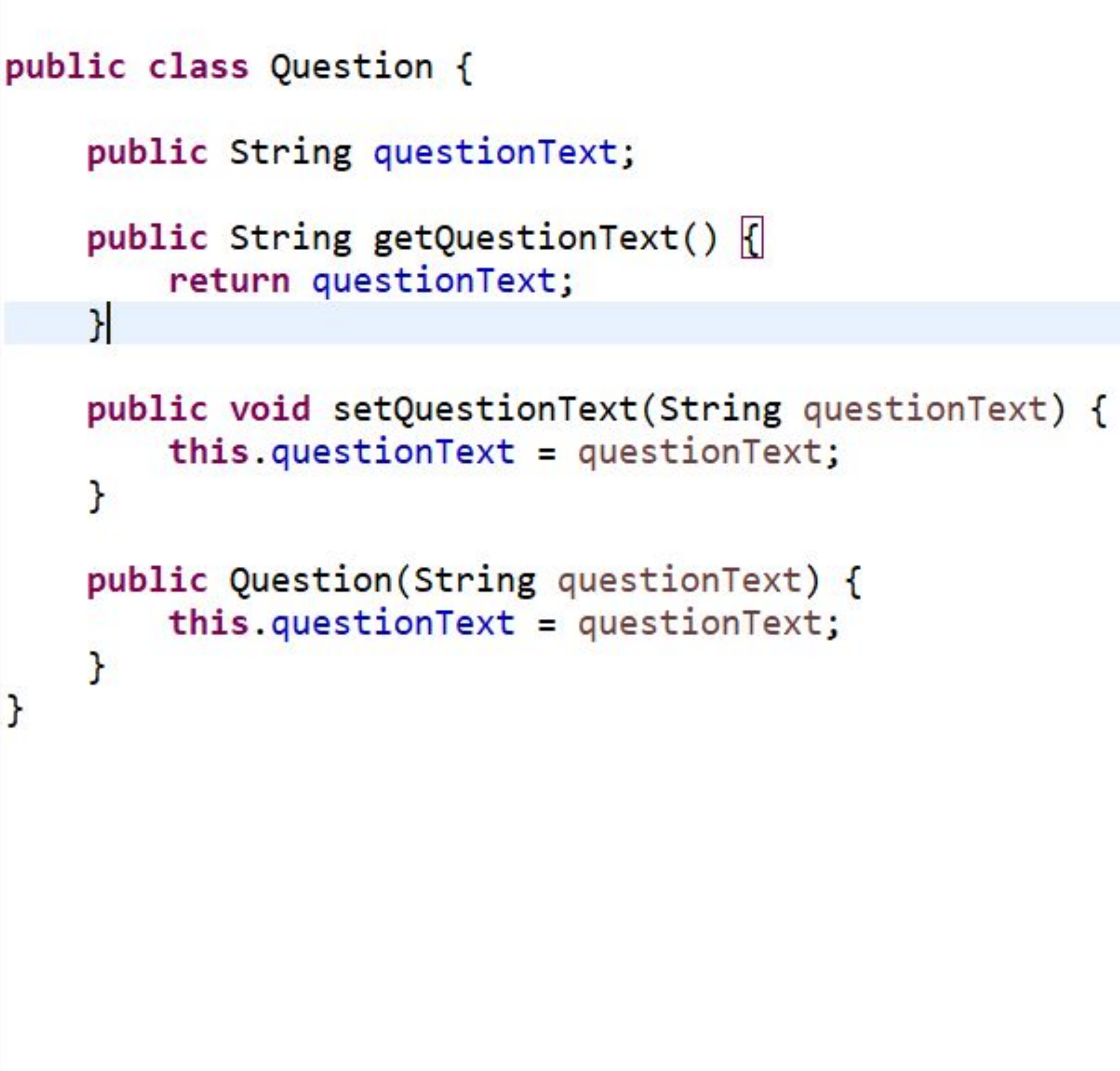
S	Single Responsibility Principle
O	Open-Closed Principle
L	Liskov Substitution Principle
I	Interface Segregation Principle
D	Dependency Inversion Principle

Single Responsibility Principle

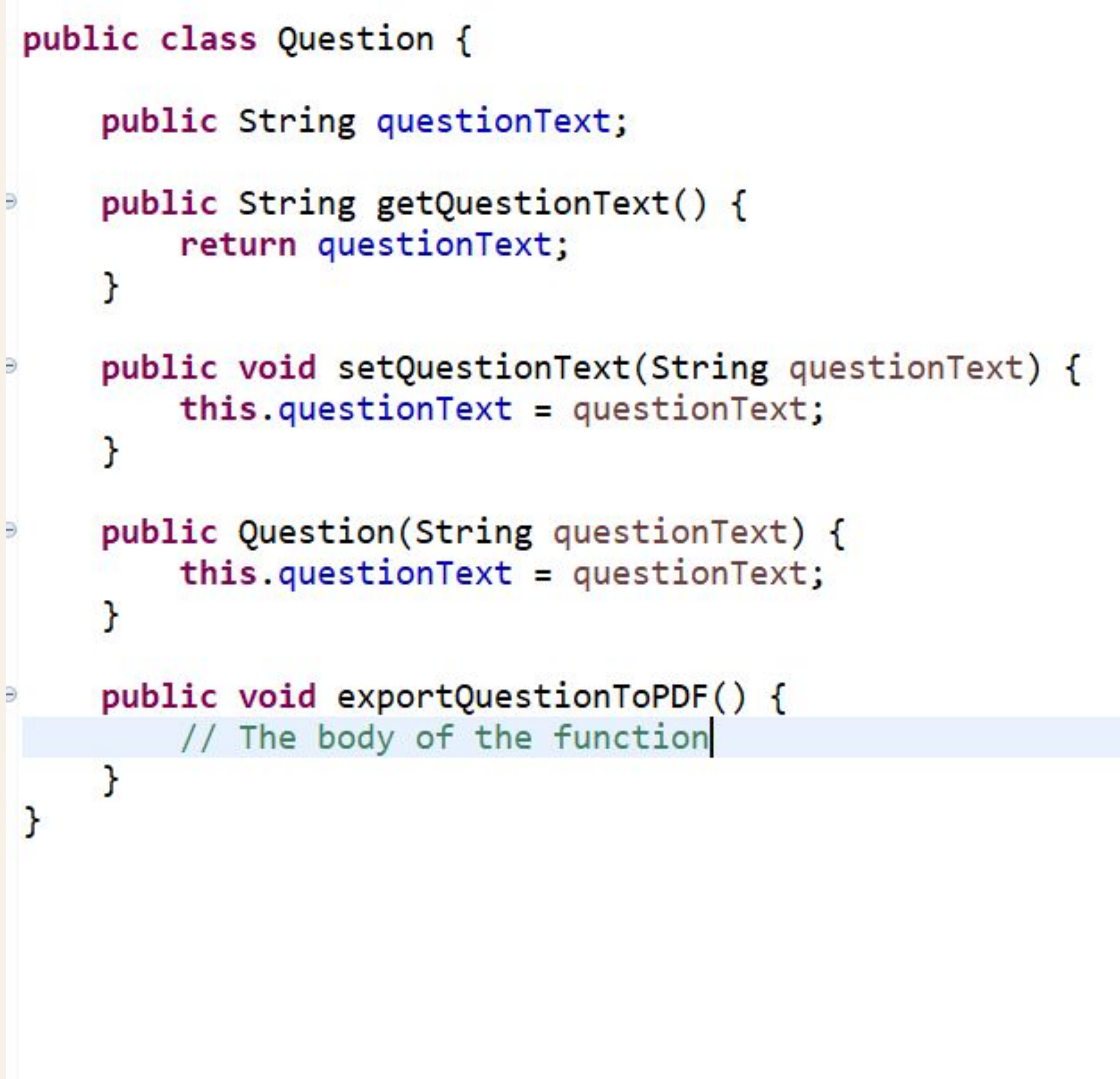


Classes should have a **single responsibility** – a class shouldn't **change for more than one reason.**

Single Responsibility Principle



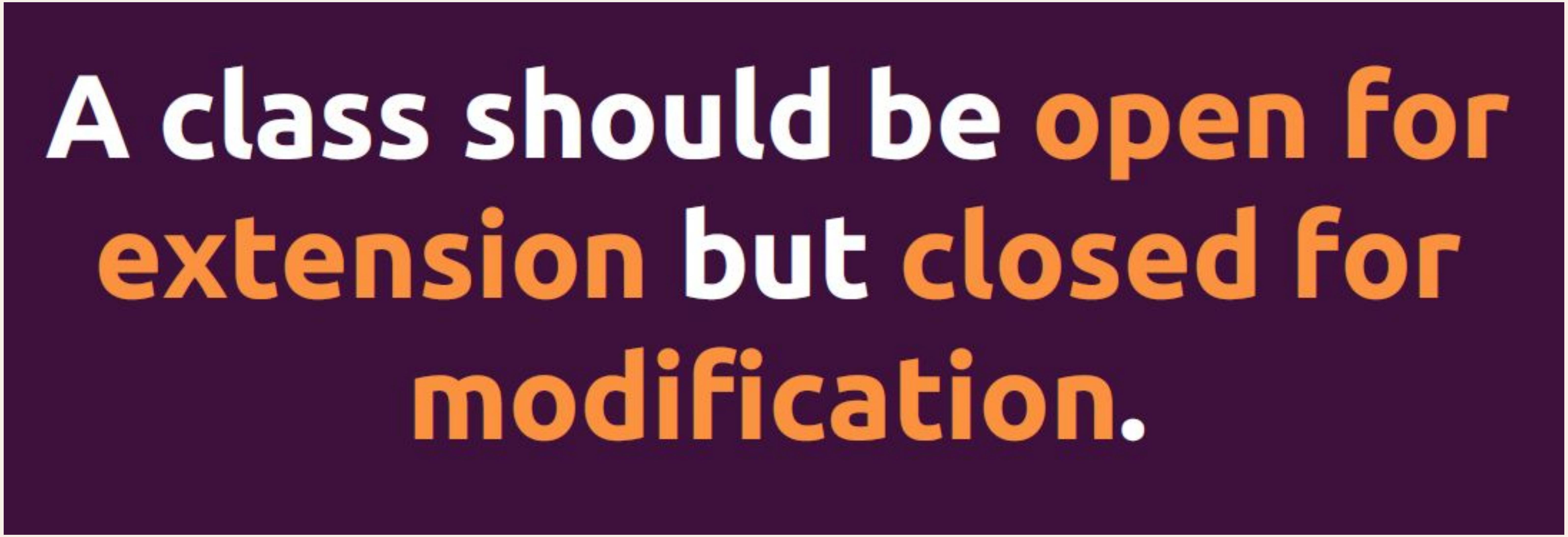
```java
public class Question {

    public String questionText;

    public String getQuestionText() {
        return questionText;
    }

    public void setQuestionText(String questionText) {
        this.questionText = questionText;
    }

    public Question(String questionText) {
        this.questionText = questionText;
    }
}
```



```java
public class Question {

    public String questionText;

    public String getQuestionText() {
        return questionText;
    }

    public void setQuestionText(String questionText) {
        this.questionText = questionText;
    }

    public Question(String questionText) {
        this.questionText = questionText;
    }

    public void exportQuestionToPDF() {
        // The body of the function
    }
}
```

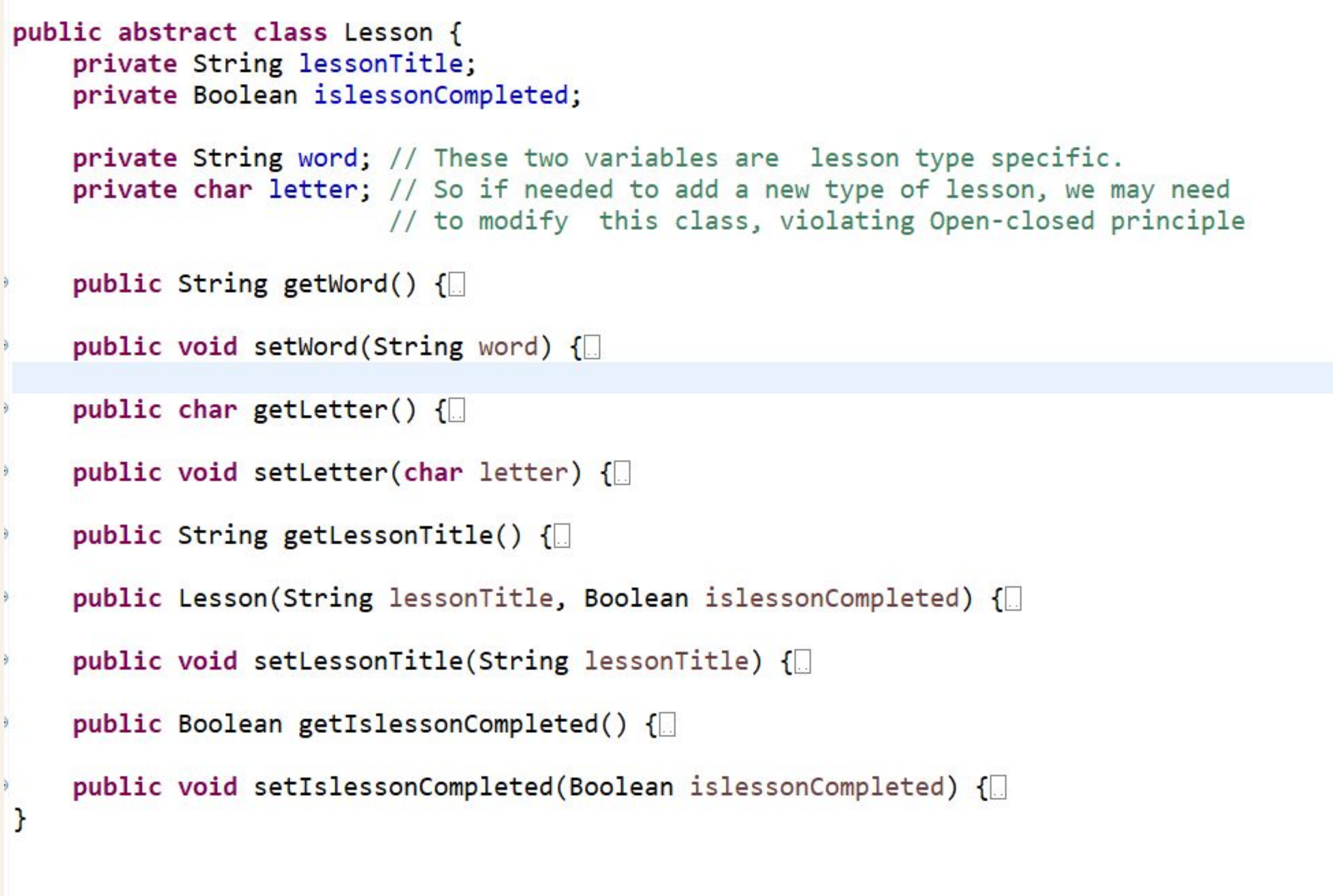
Picture 2 have a function which is not directly related to its responsibility

Open Closed Principle



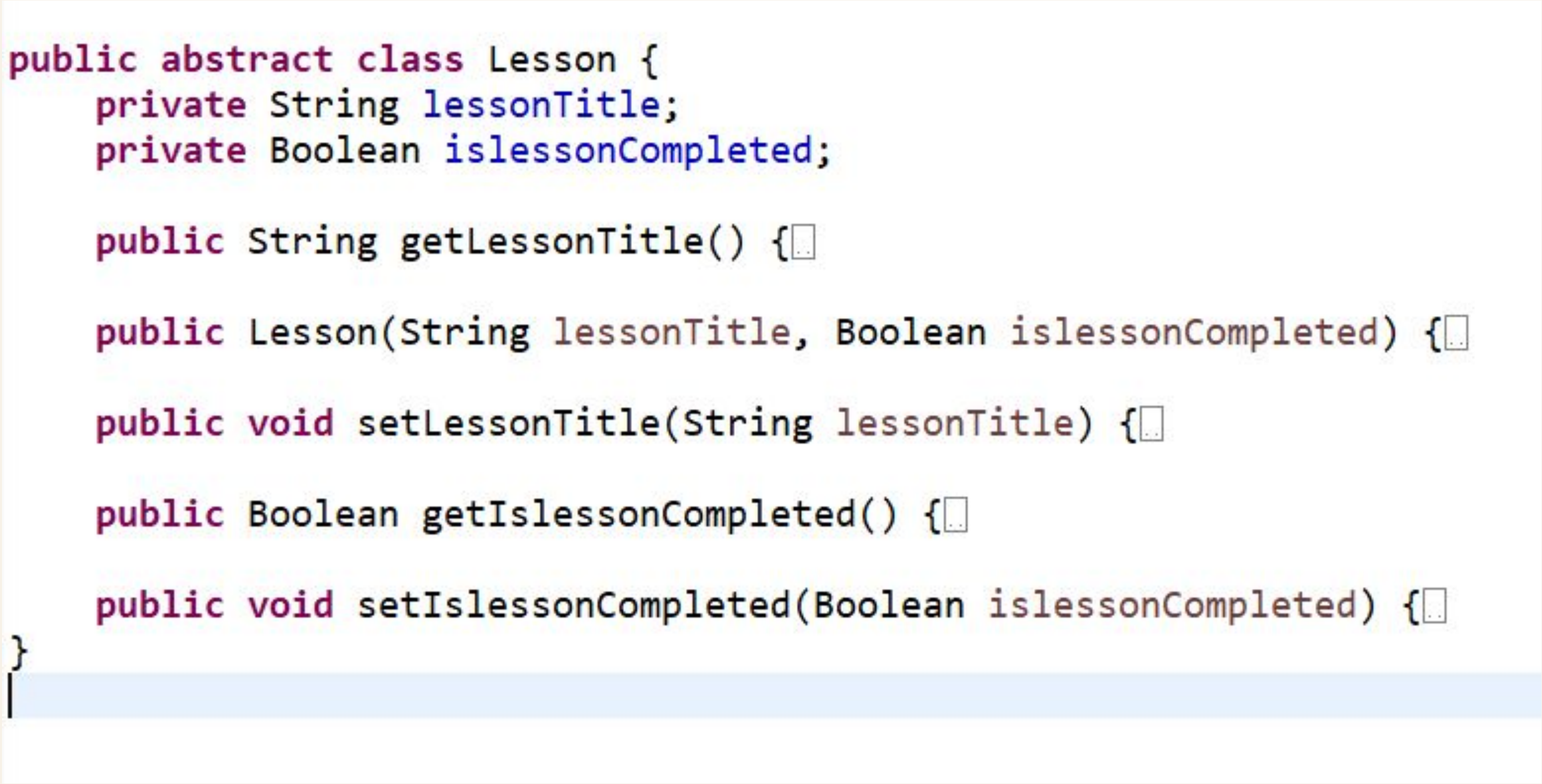
A class should be open for extension but closed for modification.

Open Closed Principle



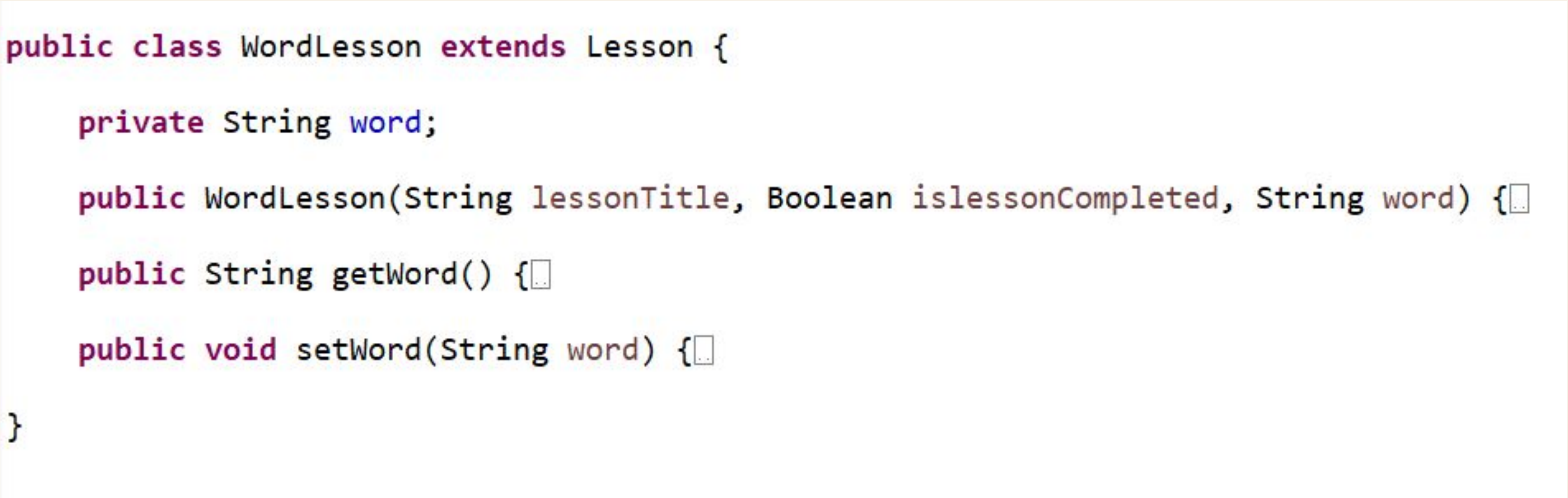
```
public abstract class Lesson {
    private String lessonTitle;
    private Boolean islessonCompleted;

    private String word; // These two variables are  lesson type specific.
    private char letter; // So if needed to add a new type of lesson, we may need
                         // to modify  this class, violating Open-closed principle

    public String getWord() {□

    public void setWord(String word) {□

    public char getLetter() {□

    public void setLetter(char letter) {□

    public String getLessonTitle() {□

    public Lesson(String lessonTitle, Boolean islessonCompleted) {□

    public void setLessonTitle(String lessonTitle) {□

    public Boolean getIslessonCompleted() {□

    public void setIslessonCompleted(Boolean islessonCompleted) {□
}
```

Open Closed Principle

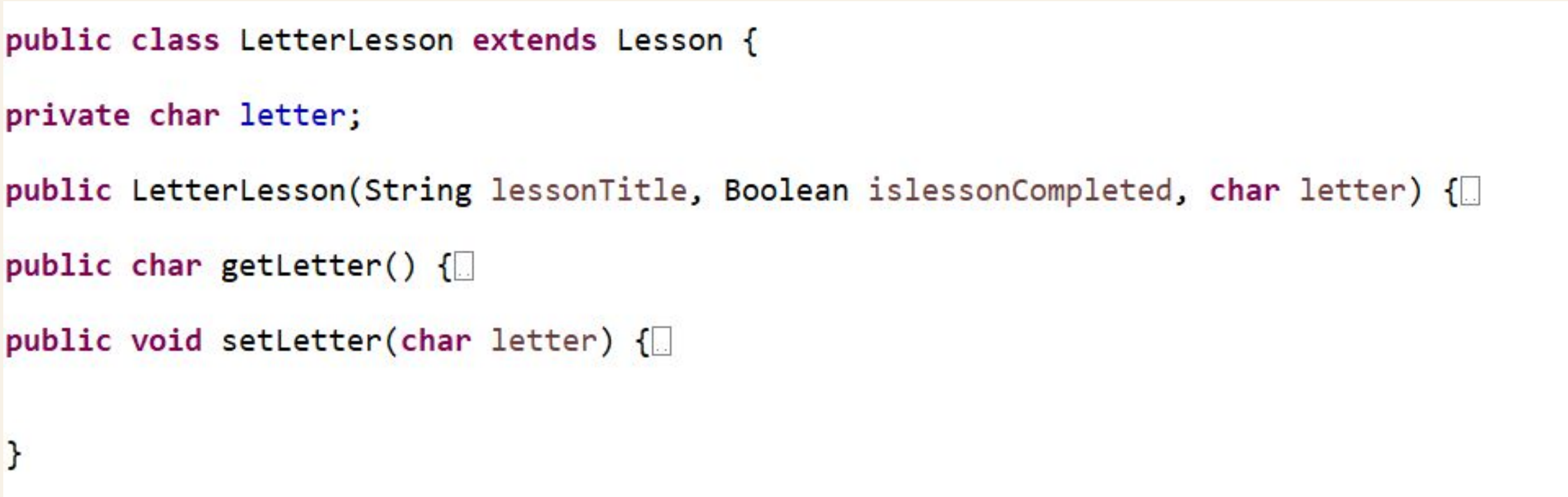


```
public abstract class Lesson {
    private String lessonTitle;
    private Boolean islessonCompleted;

    public String getLessonTitle() {□

    public Lesson(String lessonTitle, Boolean islessonCompleted) {□

    public void setLessonTitle(String lessonTitle) {□

    public Boolean getIslessonCompleted() {□

    public void setIslessonCompleted(Boolean islessonCompleted) {□
}
```

Here, addition of a new type of lesson doesnt need to modify Lesson class, just need to only extend

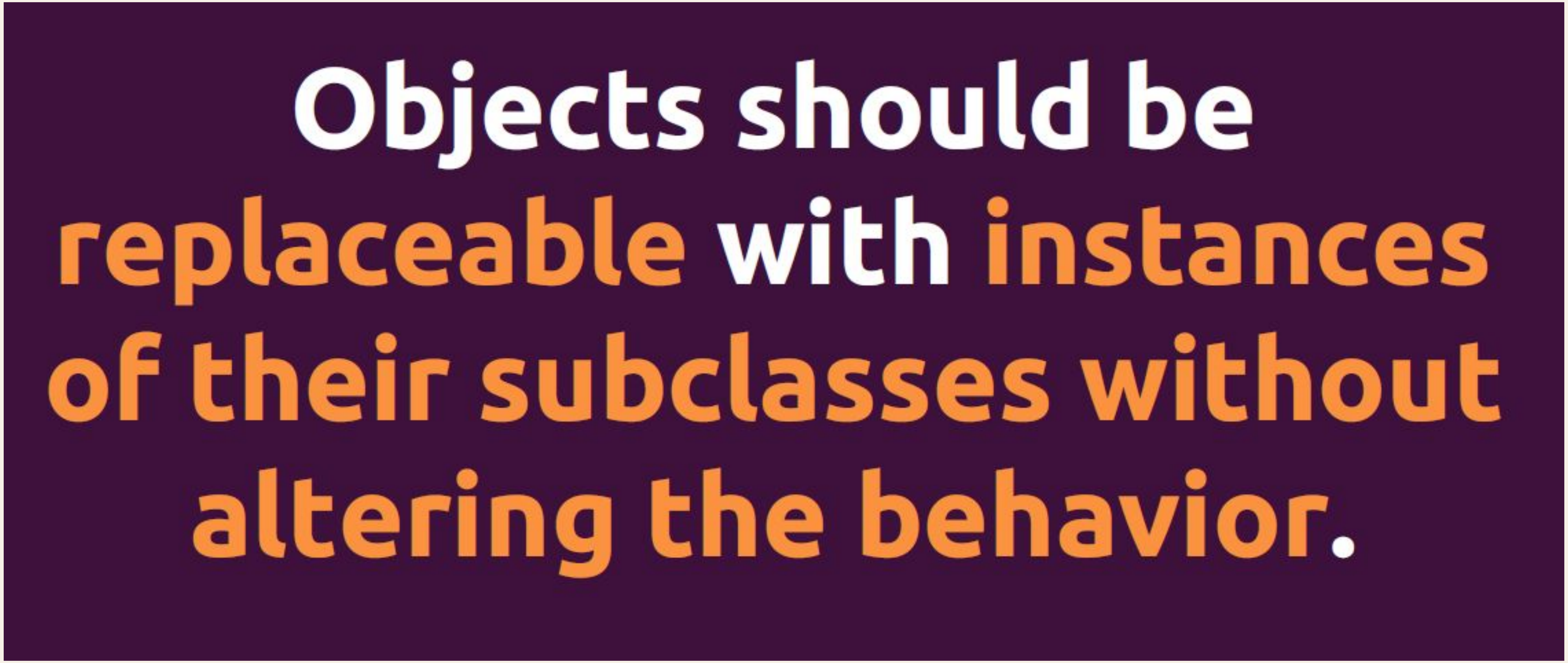


```
public class WordLesson extends Lesson {

    private String word;

    public WordLesson(String lessonTitle, Boolean islessonCompleted, String word) {□

    public String getWord() {□

    public void setWord(String word) {□

}
```



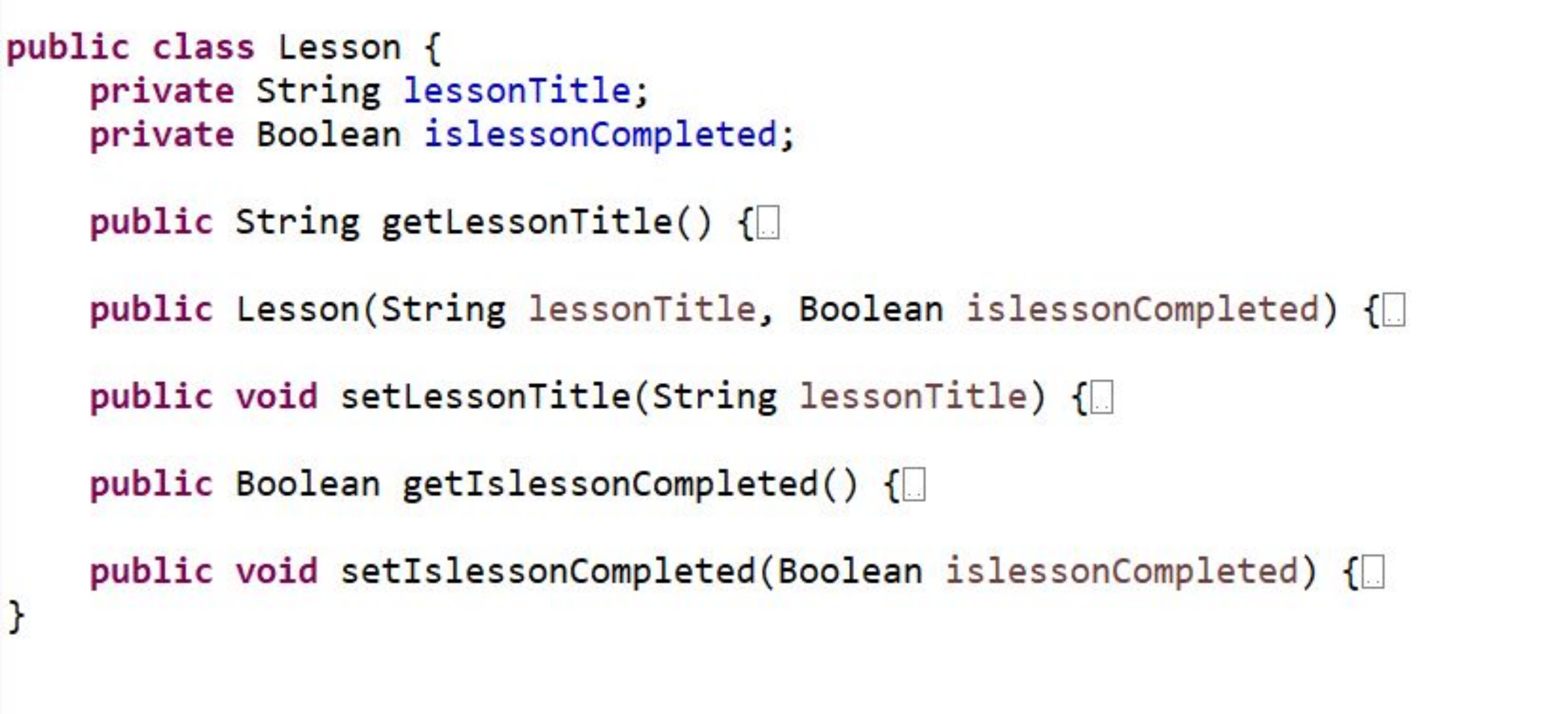
```
public class LetterLesson extends Lesson {

private char letter;

public LetterLesson(String lessonTitle, Boolean islessonCompleted, char letter) {□

public char getLetter() {□

public void setLetter(char letter) {□


}
```

Liskov Substitution Principle



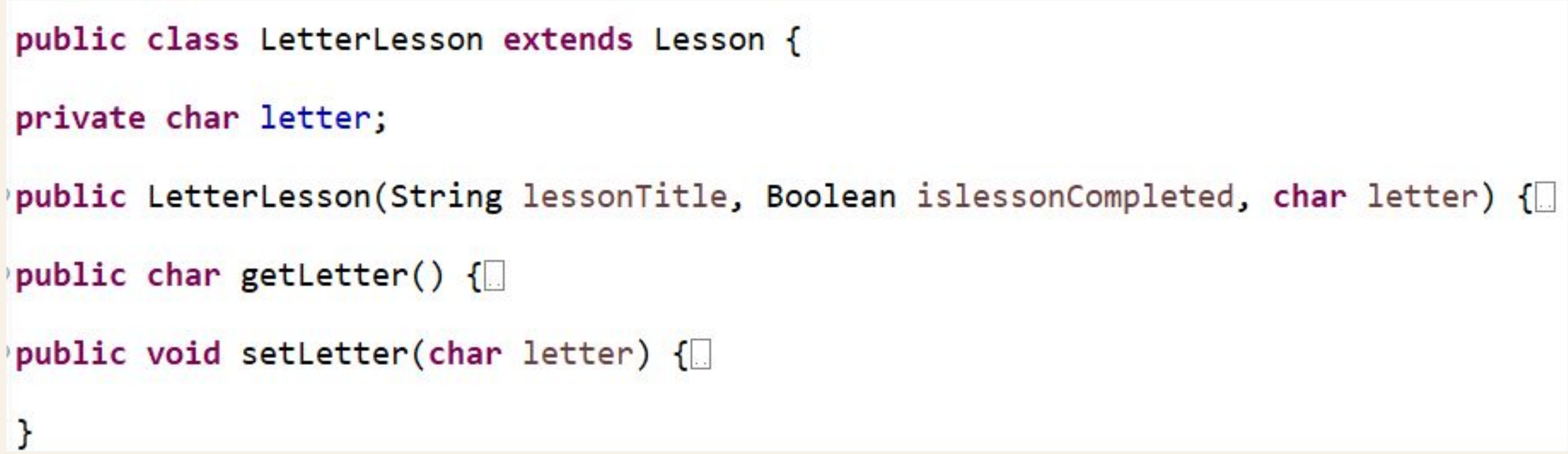
Objects should be replaceable with instances of their subclasses without altering the behavior.

Liskov Substitution Principle

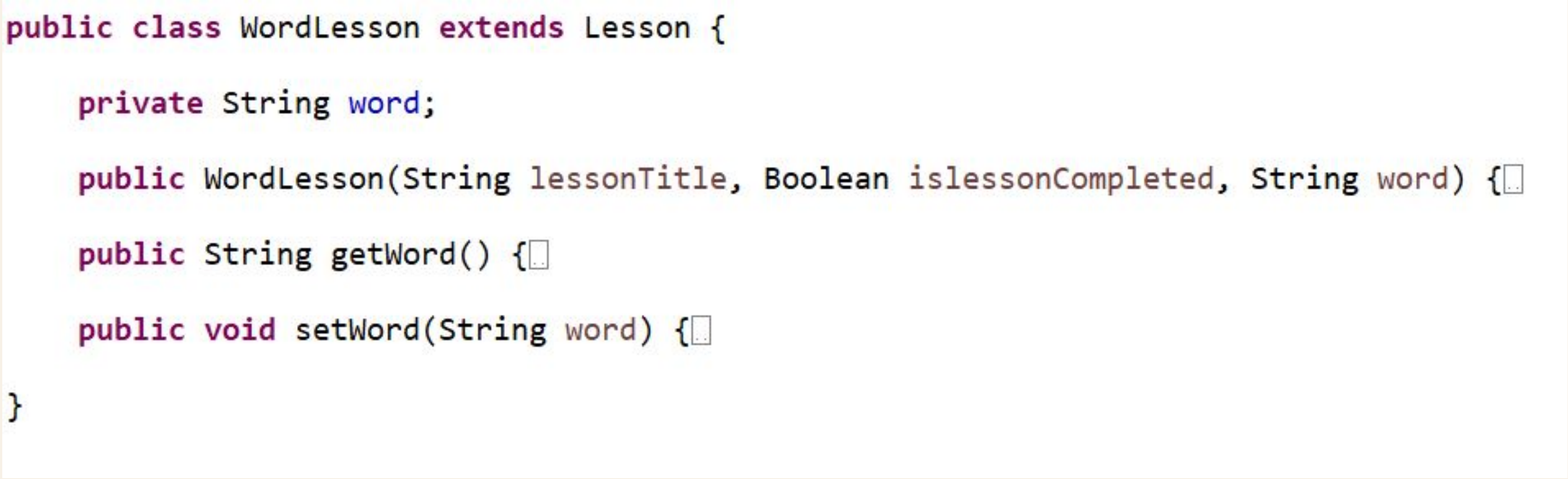


```java
public class Lesson {
    private String lessonTitle;
    private Boolean islessonCompleted;

    public String getLessonTitle() {□

    public Lesson(String lessonTitle, Boolean islessonCompleted) {□

    public void setLessonTitle(String lessonTitle) {□

    public Boolean getIslessonCompleted() {□

    public void setIslessonCompleted(Boolean islessonCompleted) {□
}
```

Here, wordLesson and LetterLesson are the child classes of Lesson.

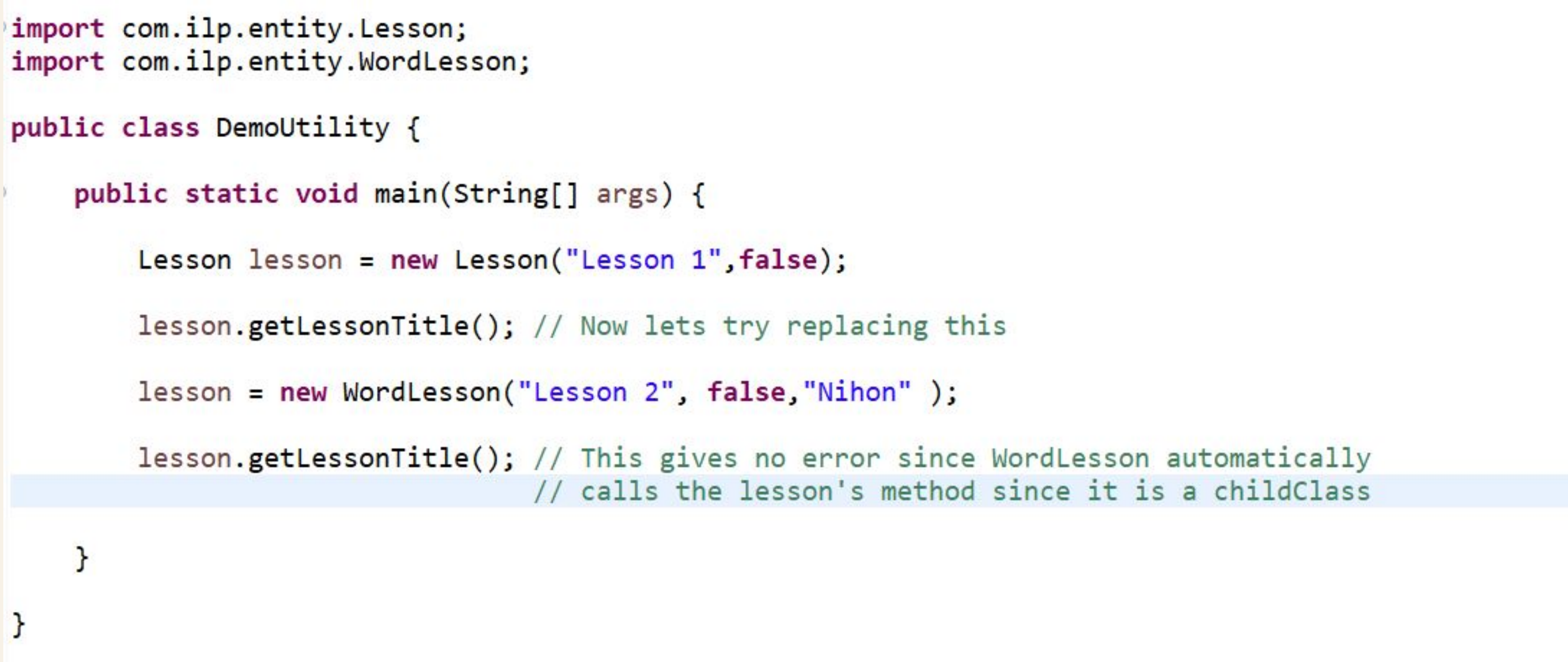


```java
public class LetterLesson extends Lesson {

private char letter;

public LetterLesson(String lessonTitle, Boolean islessonCompleted, char letter) {□

public char getLetter() {□

public void setLetter(char letter) {□

}
```



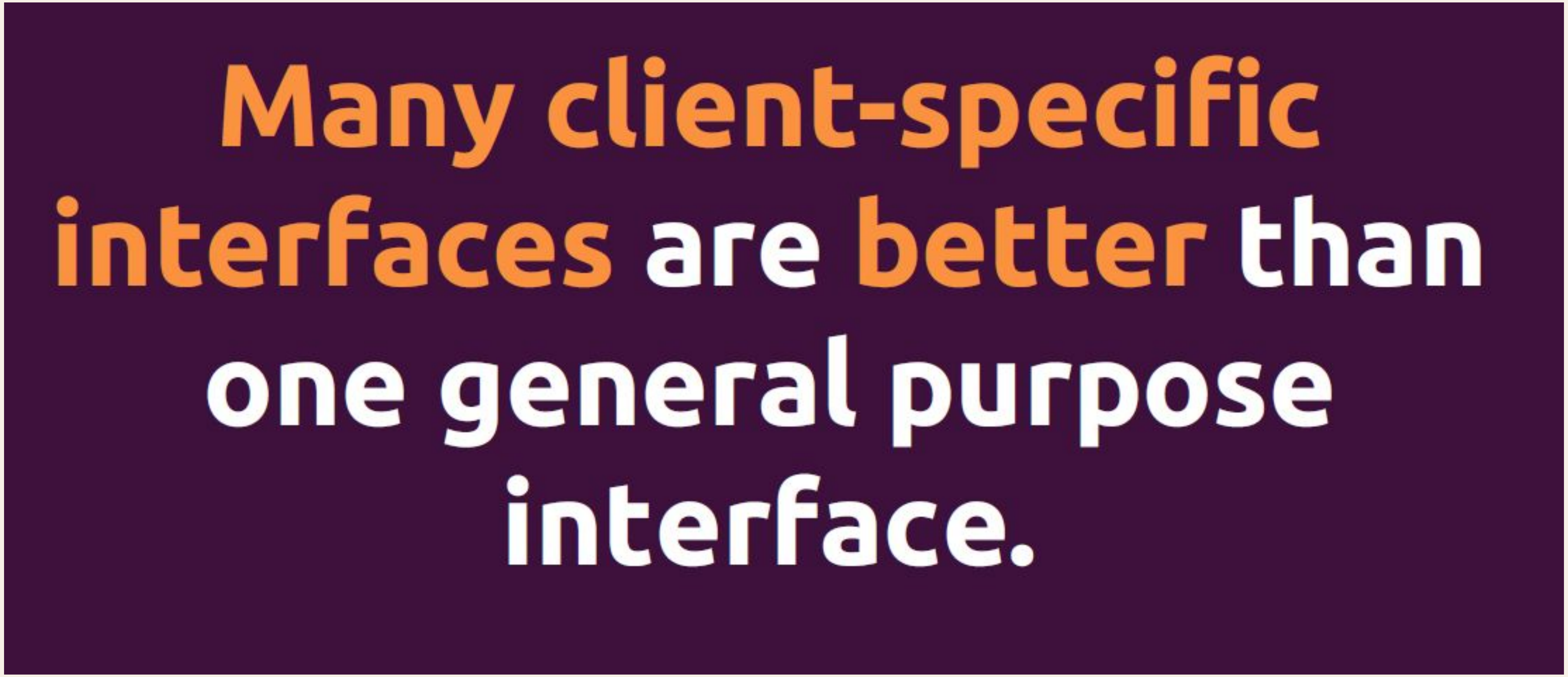
```java
public class WordLesson extends Lesson {

    private String word;

    public WordLesson(String lessonTitle, Boolean islessonCompleted, String word) {□

    public String getWord() {□

    public void setWord(String word) {□

}
```

Liskov Substitution Principle



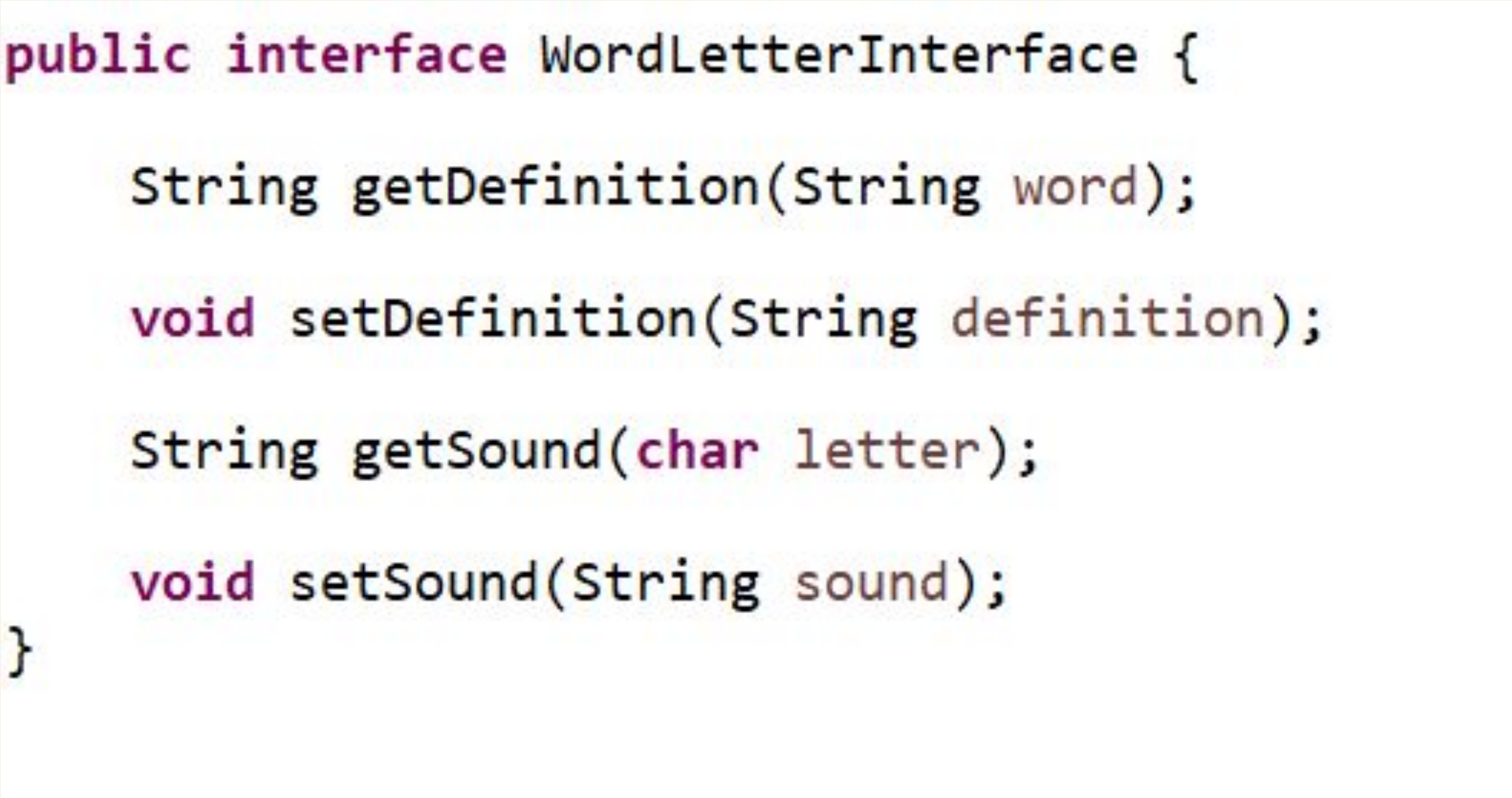
```
import com.ilp.entity.Lesson;
import com.ilp.entity.WordLesson;

public class DemoUtility {

    public static void main(String[] args) {

        Lesson lesson = new Lesson("Lesson 1",false);

        lesson.getLessonTitle(); // Now lets try replacing this

        lesson = new WordLesson("Lesson 2", false,"Nihon" );

        lesson.getLessonTitle(); // This gives no error since WordLesson automatically
                                 // calls the lesson's method since it is a childClass

    }

}
```

Interface Segregation Principle



Many client-specific interfaces are better than one general purpose interface.

Interface Segregation Principle

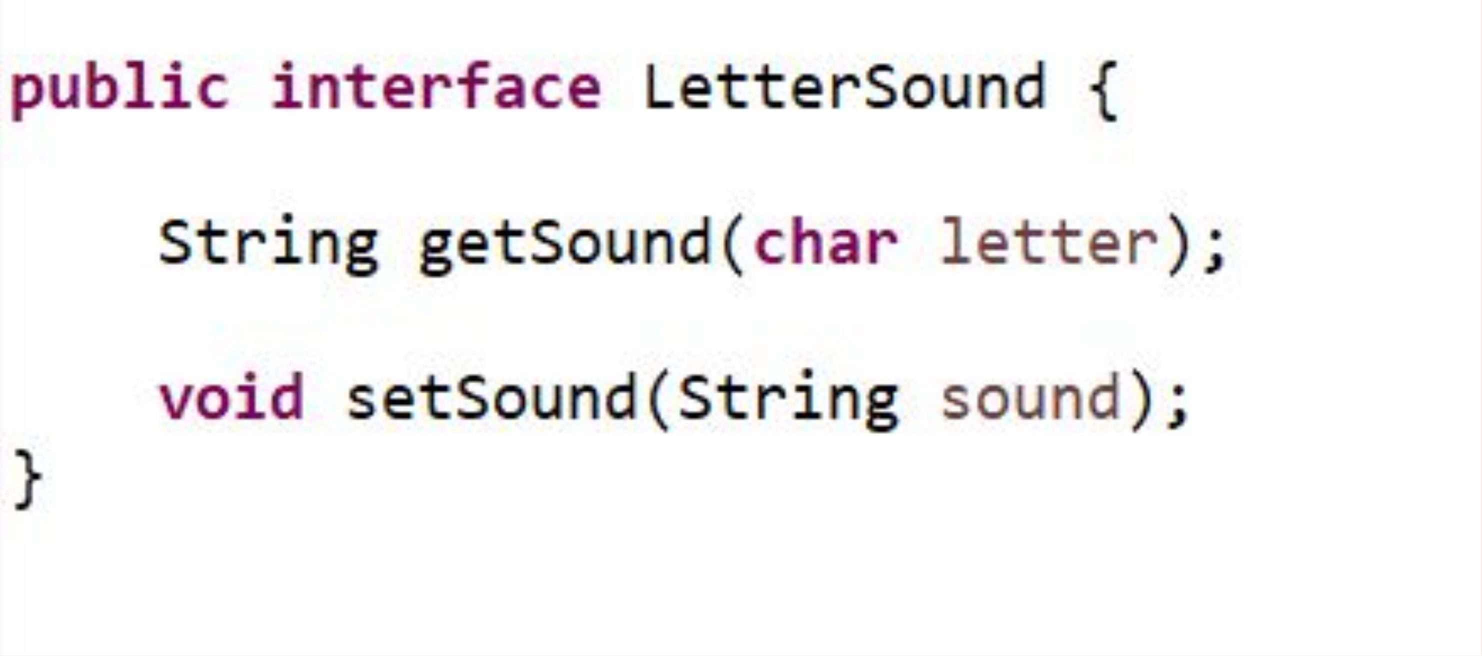


```java
public interface WordLetterInterface {

    String getDefinition(String word);

    void setDefinition(String definition);

    String getSound(char letter);

    void setSound(String sound);
}
```

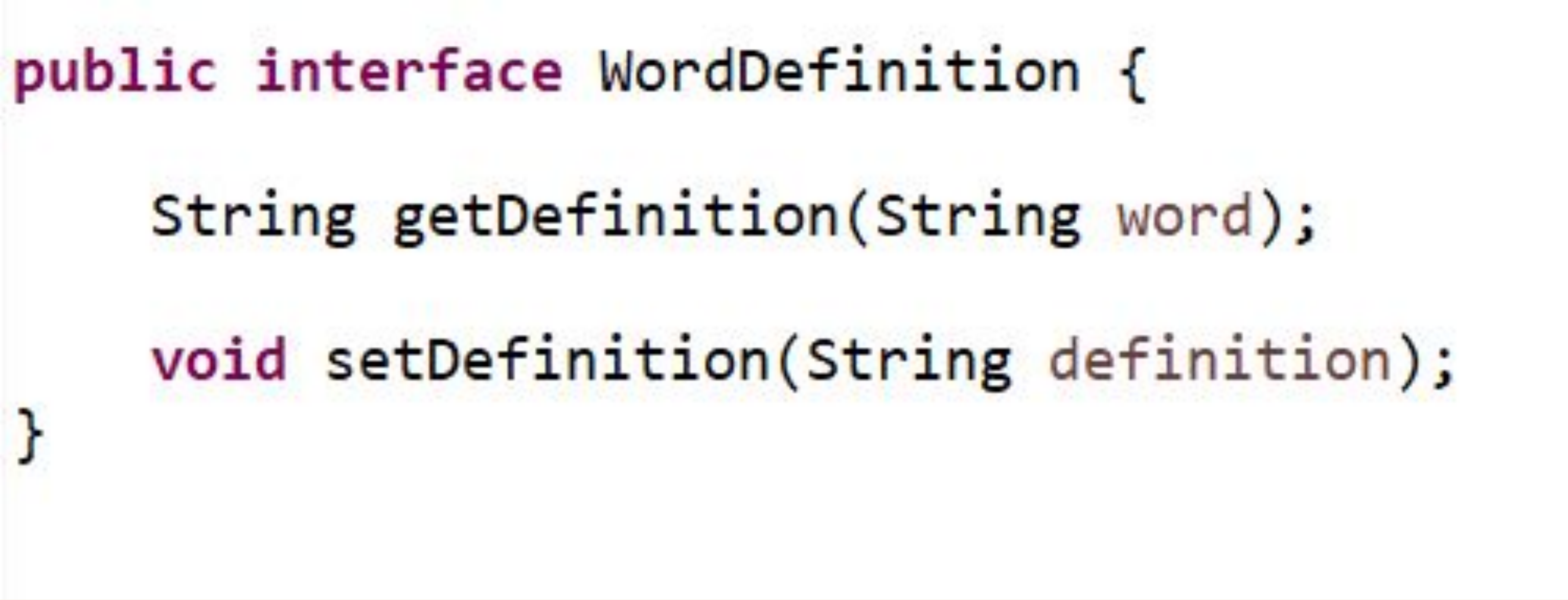
The definition related functions and sound related functions are for different classes.

Writing both in a single interface forces the implementaton of unwanted functions

Interface Segregation Principle



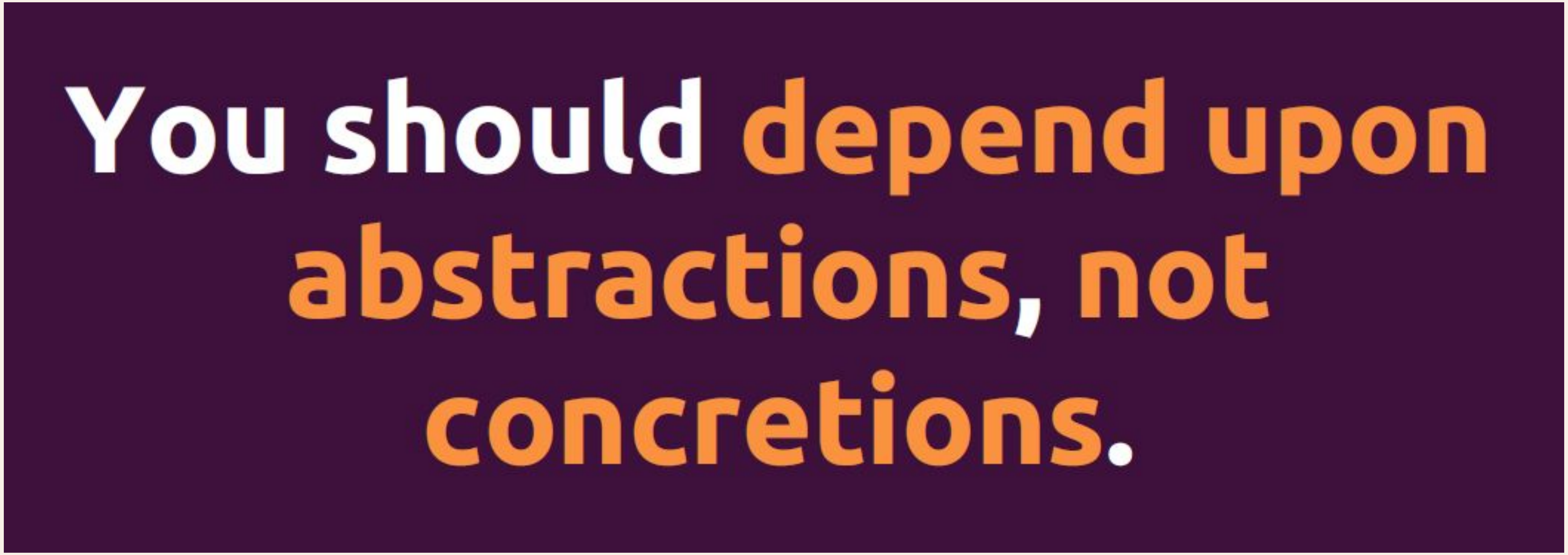
```java
public interface LetterSound {

    String getSound(char letter);

    void setSound(String sound);
}
```



```java
public interface WordDefinition {

    String getDefinition(String word);

    void setDefinition(String definition);
}
```

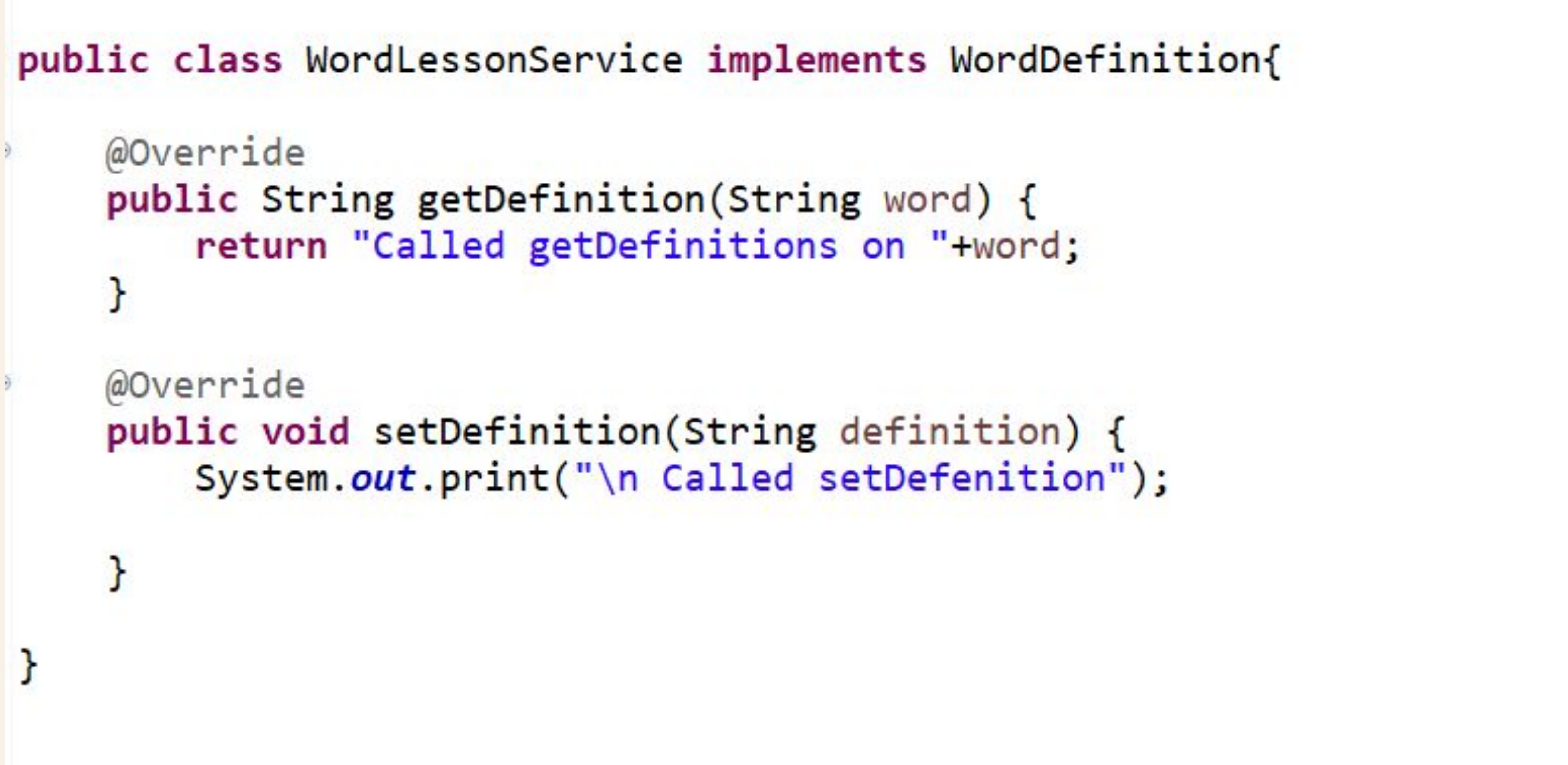
By separating them into 2 interfaces, we only have to implemented the needed function definitions.

Dependency Inversion Principle



You should depend upon abstractions, not concretions.

Dependency Inversion Principle



```
public class WordLessonService implements WordDefinition{

    @Override
    public String getDefinition(String word) {
        return "Called getDefinitions on "+word;
    }

    @Override
    public void setDefinition(String definition) {
        System.out.print("\n Called setDefenition");

    }

}
```

Here , instead of relying on a low level module, the module depends on abstraction

THANK YOU