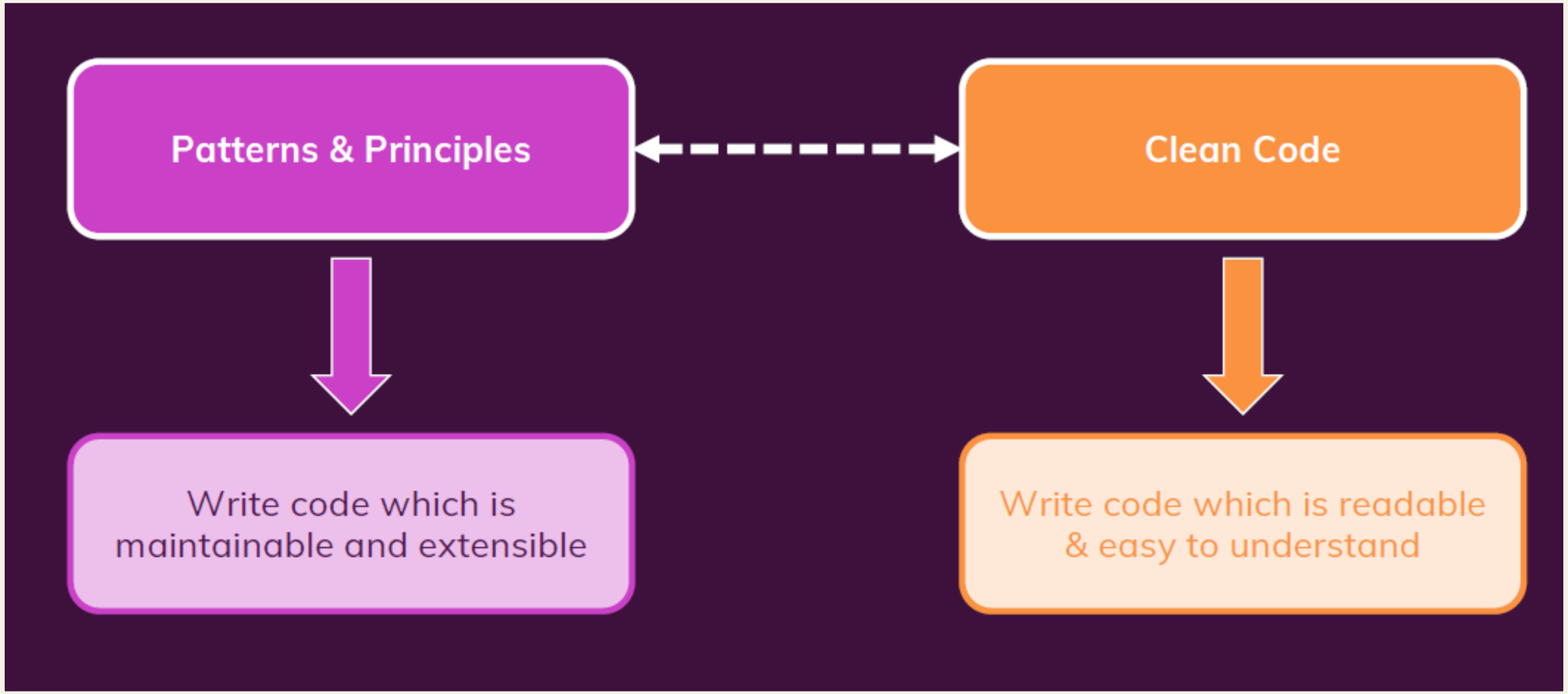
SOLID Principles

Aloyise Biju Mathew
Trainee – ILP Batch 2

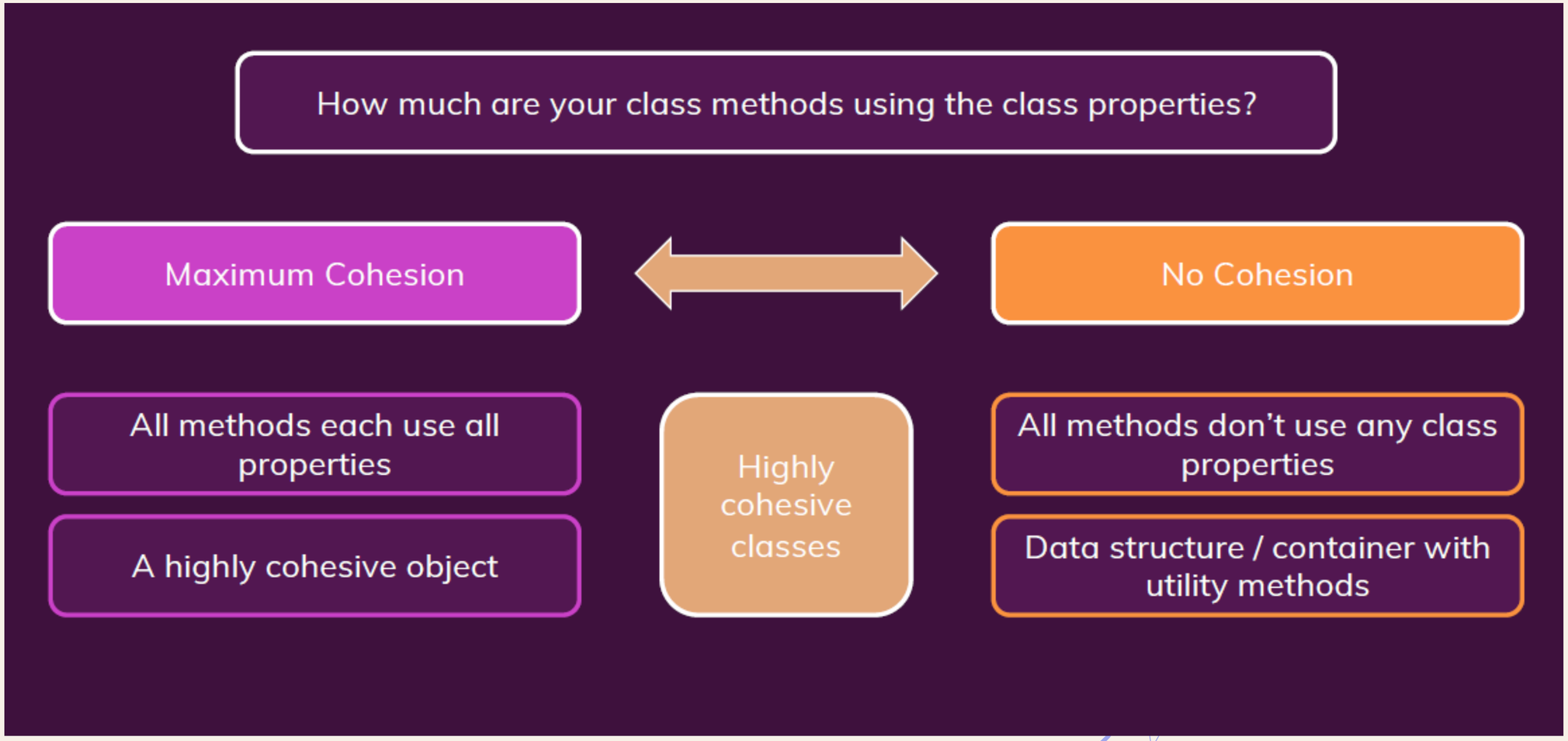
Applying SOLID Principles to Duolingo – Questions Section

Clean Code and Principle Patterns

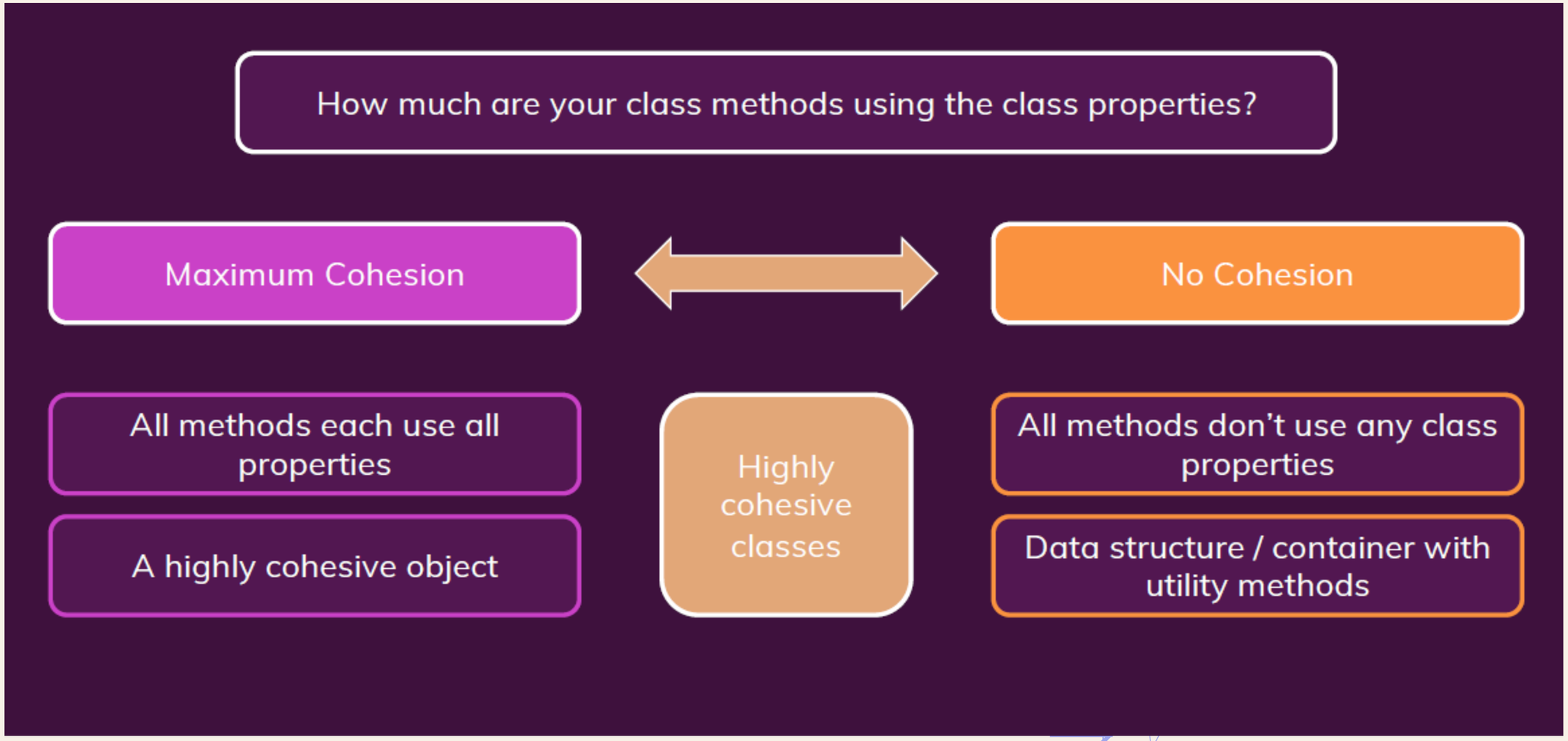


Classes should be small

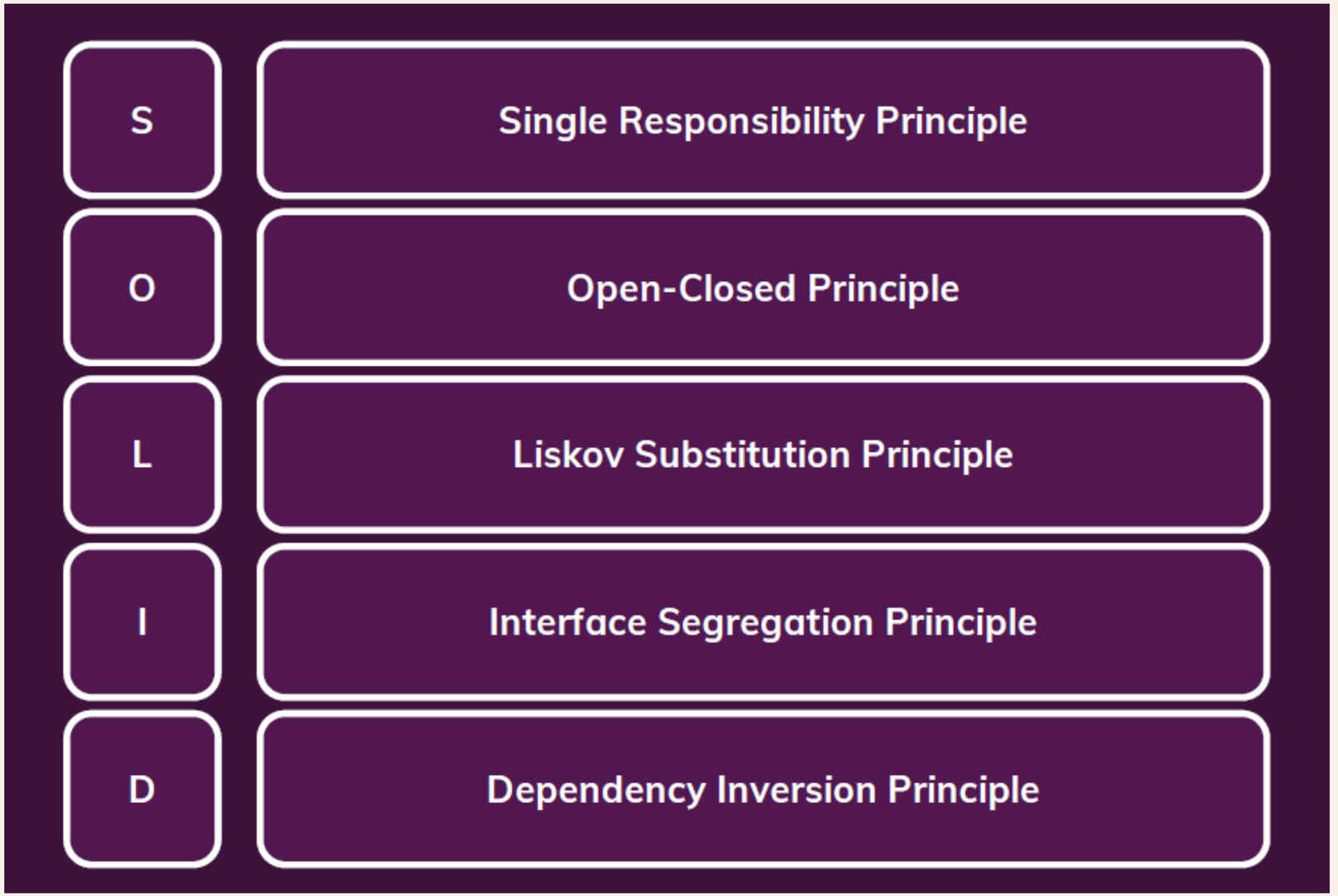
Cohesion


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



Single Responsibility Principle

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

Single Responsibility Principle

```java
import com.ilp.interfaces.Option;

public class OptionText implements Option {
    private String optionText;

    public OptionText(String optionText) {
        this.optionText = optionText;
    }

    @Override
    public String getText() {
        return optionText;
    }

    public void setText(String optionText) {
        this.optionText = optionText;
    }
}
```

```java
public class OptionImage extends OptionText {
    private String imageUrl;

    public OptionImage(String optionContent, String imageUrl) {
        super(optionContent);
        this.imageUrl = imageUrl;
    }

    public String getImageUrl() {
        return imageUrl;
    }

    public void setImageUrl(String imageUrl) {
        this.imageUrl = imageUrl;
    }
}
```

Each class has a single responsibility.

OptionText is responsible for representing text options, and **OptionImage** is responsible for representing image options.

Open Closed Principle

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

Open Closed Principle

```java
public class QuestionBase {
    private String questionText;
    private int correctIndex;

    public QuestionBase(String questionText, int correctIndex) {
        this.questionText = questionText;
        this.correctIndex = correctIndex;
    }
    //...
```

```java
public class QuestionWithTextOptions extends QuestionBase {
    private List<OptionText> options;

    public QuestionWithTextOptions(String questionText, int correctIndex, List<OptionText> options) {
        super(questionText, correctIndex);
        this.options = options;
    }

    public List<OptionText> getOptions() {
        return options;
    }

    public void setOptions(List<OptionText> options) {
        this.options = options;
    }
}
```

Open Closed Principle

```java
public class QuestionWithImageOptions extends QuestionBase {
    private List<OptionImage> options;

    public QuestionWithImageOptions(String questionText, int correctIndex, List<OptionImage> options) {
        super(questionText, correctIndex);
        this.options = options;
    }

    public List<OptionImage> getOptions() {
        return options;
    }

    public void setOptions(List<OptionImage> options) {
        this.options = options;
    }
}
```

The base **QuestionBase** class is open for extension as new types of questions can be created by extending it (e.g., **QuestionWithImageOptions**, **QuestionWithTextOptions**).
It's closed for modification, as existing code doesn't need to change when introducing new question types.

Liskov Substitution Principle

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

Liskov Substitution Principle

```java
public class ChallengeAssist extends QuestionWithTextOptions {
    public ChallengeAssist(String questionText, int correctIndex, List<OptionText> options) {
        super(questionText, correctIndex, options);
    }
}
```

```java
public class ChallengeSelect extends QuestionWithImageOptions {
    public ChallengeSelect(String questionText, int correctIndex, List<OptionImage> options) {
        super(questionText, correctIndex, options);
    }
}
```

All the subclasses (**ChallengeAssist**, **ChallengeSelect**) can be substituted for their base class (**QuestionWithTextOptions** or **QuestionWithImageOptions**) without affecting the correctness of the program.

Interface Segregation Principle

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

Interface Segregation Principle

```java
public interface DisplayQuestion {
    public void displayQuestion();
}
```

```java
public interface PlayAudio {
    void playAudio(String questionText);
}
```

```java
public interface QuestionResult {
    void checkAnswer(String userAnswer);
    void skipAnswer();
}
```

The **DisplayQuestion, PlayAudio and QuestionResult** interface follows ISP as it contains methods ensuring that the implementing classes are not forced to implement unnecessary methods.

Dependency Inversion Principle

You should depend upon abstractions, not concretions.

Dependency Inversion Principle

```java
public class QuestionServiceWithAudio extends QuestionService implements PlayAudio {
    public QuestionServiceWithAudio(QuestionWithTextOptions question) {
        super(question);
    }
    @Override
    public void playAudio(String questionText) {
        System.out.println("Playing audio for: " + questionText);
    }
}
```

QuestionService and **QuestionServiceWithAudio** depend on abstractions (**QuestionResult, PlayAudio,** and **DisplayQuestion**), not on concrete implementations. This allows for flexibility and ease of extension without modifying existing code.

```java
public class QuestionService implements QuestionResult, DisplayQuestion {
    private QuestionWithTextOptions question;

    public QuestionService(QuestionWithTextOptions question) {
        this.question = question;
    }

    @Override
    public void checkAnswer(String userAnswer) {
        System.out.println("Checking answer");
    }

    @Override
    public void skipAnswer() {
        System.out.println("Skipping answer Checking");
    }

    @Override
    public void displayQuestion() {
        System.out.println(question.getQuestionText());
    }

}
```

Dependency Inversion Principle

```java
public interface PlayAudio {
    void playAudio(String questionText);
}
```

```java
public interface DisplayQuestion {
    public void displayQuestion();
}
```

```java
public interface QuestionResult {
    void checkAnswer(String userAnswer);
    void skipAnswer();
}
```

QuestionResult, PlayAudio, and DisplayQuestion Interfaces represent abstractions that high-level modules (e.g., **QuestionService** and **QuestionServiceWithAudio**) depend on. They don't depend on details but on abstractions.