Project Title: Technical Documentation for QuizApplication – A Detailed Overview

Jian Gong - 8956828

Prayerson Chauhan - 8878082

Shrirang Raval - 8958825

Yash Patel - 9049539

Yash Shah - 8990493

High-Quality Software Programming

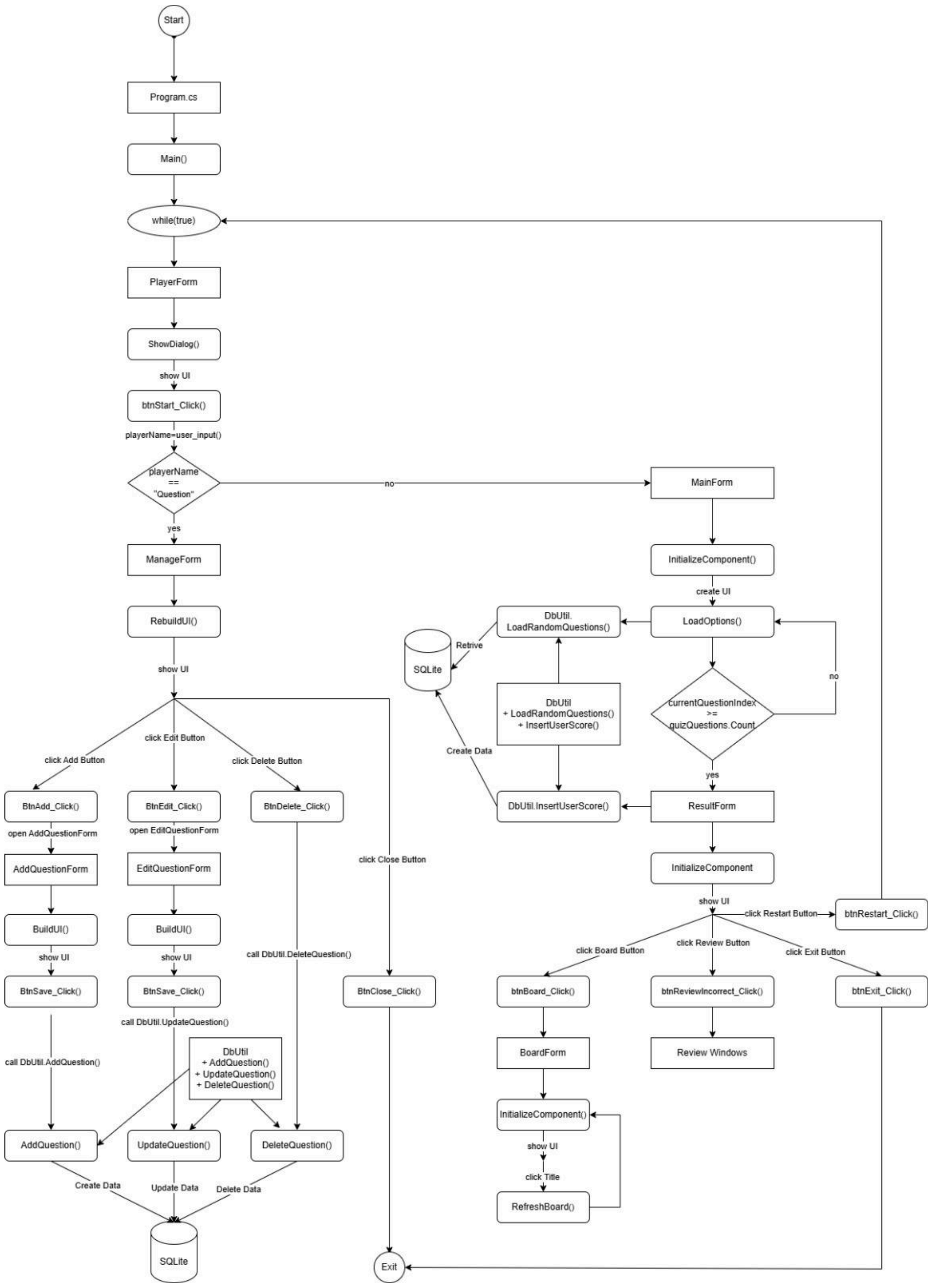
PROG8051 - Winter 2025 - Section 1

Shankar Iyer

30th March 2025

Project Overview

The QuizApplication is a thorough C# Windows Forms application built with.NET 8.0. It provides visitors with a fully interactive quiz experience by supporting several question formats such as multiple-choice (single answer), checkbox-based (many responses), and text input. The program has a player interface, a result analysis module, and an administrator dashboard for controlling quiz content. It also uses SQLite for local data storage and provides graphic components to increase user engagement.


Start
Program.cs
Main()
while(true)
PlayerForm
ShowDialog()
show UI
btnStart_Click()
playerName=user_input()
playerName == "Question"
no
yes
ManageForm
RebuildUI()
show UI
click Add Button
click Edit Button
click Delete Button
click Close Button
BtnAdd_Click()
open AddQuestionForm
AddQuestionForm
BuildUI()
show UI
BtnSave_Click()
call DbUtil.AddQuestion()
BtnEdit_Click()
open EditQuestionForm
EditQuestionForm
BuildUI()
show UI
BtnSave_Click()
call DbUtil.UpdateQuestion()
BtnDelete_Click()
call DbUtil.DeleteQuestion()
BtnClose_Click()
DbUtil + AddQuestion() + UpdateQuestion() + DeleteQuestion()
AddQuestion()
UpdateQuestion()
DeleteQuestion()
Create Data
Update Data
Delete Data
SQLite
Exit
MainForm
InitializeComponent()
create UI
LoadOptions()
DbUtil. LoadRandomQuestions()
Retrive
SQLite
DbUtil + LoadRandomQuestions() + InsertUserScore()
Create Data
currentQuestionIndex >= quizQuestions.Count
no
yes
ResultForm
DbUtil.InsertUserScore()
InitializeComponent
show UI
click Restart Button
btnRestart_Click()
click Board Button
click Review Button
click Exit Button
btnBoard_Click()
btnReviewIncorrect_Click()
btnExit_Click()
BoardForm
Review Windows
InitializeComponent()
show UI
click Title
RefreshBoard()

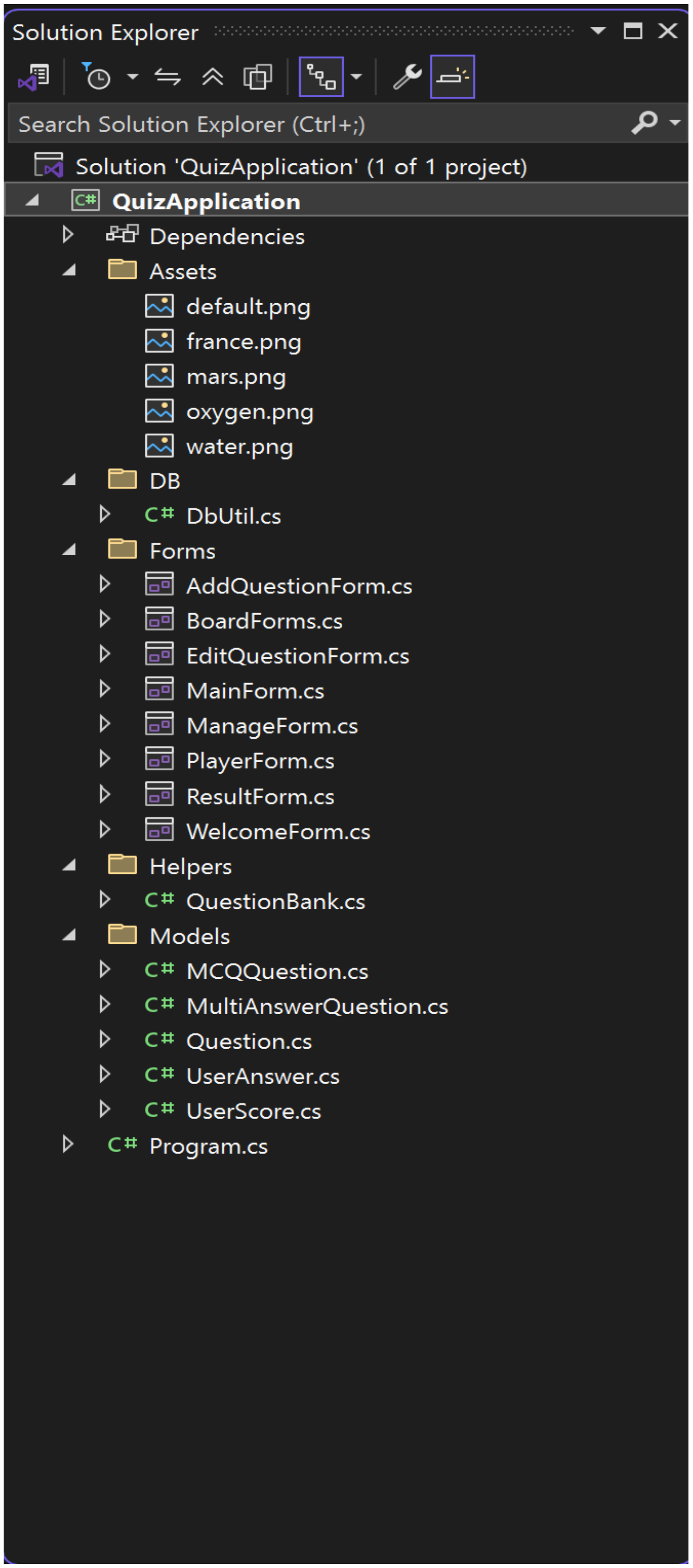
Architecture & Components

Project Structure:

```
QuizApplication/
├── Assets/                  # Image files used in quiz questions (e.g.,
france.png, water.png)
├── bin/                     # Binary output folder (generated by build)
├── DB/                      # Database utility scripts (e.g., DbUtil.cs)
├── Forms/                   # Windows Forms (UI) like MainForm,
ResultForm, etc.
│   ├── MainForm.cs
│   ├── MainForm.Designer.cs
│   ├── ResultForm.cs
│   ├── ResultForm.Designer.cs
│   └── ...
├── Helpers/                 # Helper classes for loading and managing
questions
│   └── QuestionBank.cs
├── Models/                  # Base and derived question types, user
score, and answers
│   ├── Question.cs
│   ├── MCQQuestion.cs
│   ├── MultiAnswerQuestion.cs
│   ├── UserAnswer.cs
│   └── UserScore.cs
├── obj/                     # Temporary object files (auto-generated)
├── Program.cs               # Main entry point for the application
├── QuizApplication.csproj   # C# project configuration file
└── README.md                # (If added) Documentation for the project
```

Key Concepts:

1. **Event-Driven UI**: Uses event handlers for interactivity
2. **Polymorphism**: Different question types extend a base Question class
3. **Encapsulation**: Models encapsulate data used across the application
4. **Separation of Concerns**: Forms, Models, and Helpers are logically organized



Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'QuizApplication' (1 of 1 project)
  QuizApplication
    Dependencies
    Assets
      default.png
      france.png
      mars.png
      oxygen.png
      water.png
    DB
      DbUtil.cs
    Forms
      AddQuestionForm.cs
      BoardForms.cs
      EditQuestionForm.cs
      MainForm.cs
      ManageForm.cs
      PlayerForm.cs
      ResultForm.cs
      WelcomeForm.cs
    Helpers
      QuestionBank.cs
    Models
      MCQQuestion.cs
      MultiAnswerQuestion.cs
      Question.cs
      UserAnswer.cs
      UserScore.cs
    Program.cs
```

Forms Description

`WelcomeForm`

- Starting screen of the app.
- Options include starting the quiz, managing questions, or exiting.
- Components: Buttons, Labels, Navigation events.

`PlayerForm`

- Collects the player name and moves to the quiz.
- Components: TextBox for player name, Continue button.

`MainForm`

- Main quiz interface.
- Dynamically renders questions depending on their type (radio, checkbox, text).
- Features: Countdown timer, navigation to ResultForm, real-time score tracking.
- Handles image loading for visual questions.

`ResultForm`

- Shows player name, total score, percentage, and letter grade.
- Offers a review of all questions with selected and correct answers.
- Supports a "Review Incorrect Only" feature.

`ManageForm`

- Admin panel for managing question list.
- Buttons for adding, editing, and deleting questions.

`AddQuestionForm`

- Allows admins to input new questions.
- Components: Text fields for question and options, Checkboxes for correct answers, Image uploader.

`EditQuestionForm`

- Similar to Add form, but pre-filled with selected question data.

`BoardForms`

- Optional leaderboard/history display (suggested but may need refinement).

Model Classes

```
public abstract class Question {
    public string Text { get; set; }
    public string ImagePath { get; set; }
    public abstract string GetQuestionType();
}
```

`MCQQuestion.cs`

- Inherits from Question
- Includes: Options (List), CorrectOption (string)

`MultiAnswerQuestion.cs`

- Inherits from Question
- Includes: Options (List), CorrectOptions (List)

`UserAnswer.cs`

- Stores player's answer: selected options and question reference.

`UserScore.cs`

- Stores final score, percentage, and grading logic.

Helper Classes

`QuestionBank.cs`

- Static class for holding and managing questions.
- Responsibilities:
  - Load questions from database or memory
  - Shuffle questions
  - Provide access to current question

Database Layer

`DbUtil.cs`

- Connects to Quiz.db (SQLite).
- CRUD operations for Question entities.
- Manages image paths and serializes options.

Database File: Quiz.db

- Contains tables for Questions and potentially UserScores.

Assets and Resources

- Located in Assets/
- Used to visually represent quiz questions.
- Files: france.png, mars.png, oxygen.png, water.png, default.png

Forms also contain .resx files for storing UI strings and component states.

Technologies Used

- Language: C#
- Framework: .NET 8.0 (WinForms)
- UI Framework: Windows Forms Designer
- Database: SQLite (System.Data.SQLite.dll)
- IDE: Visual Studio 2022 or higher

Build & Execution Instructions

Requirements:

- Windows OS
- Visual Studio 2022+
- .NET Desktop Development workload

Steps:

- Open QuizApplication.sln in Visual Studio.
- Restore NuGet packages.
- Build the project.
- Ensure the Assets/ and Quiz.db files are in the debug folder.
- Run the application — WelcomeForm is the entry point.

Feature Summary

- Support for:
    - Multiple Choice Questions (single answer)

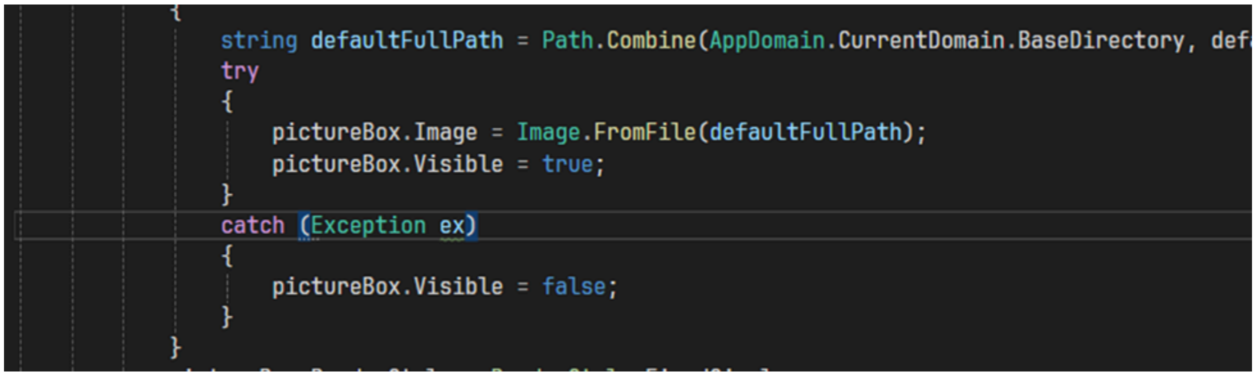
    - Checkbox Questions (multiple answers)
    - Dynamic question display
    - Result screen with grading
    - Admin panel to add/edit/delete questions
- Image support per question
- Review incorrect answers feature
- Simple SQLite backend for persistence

Potential Improvements

- Add user authentication
- Add leaderboard with persistent high scores
- Move to MVVM/MVP for separation of concerns
- Add unit tests and error handling
- Migrate to web or mobile platform for broader reach
- Export results to PDF/Excel

Code Detail Level Explanation

1. **Exception**
    a. We use try / catch when loading images.



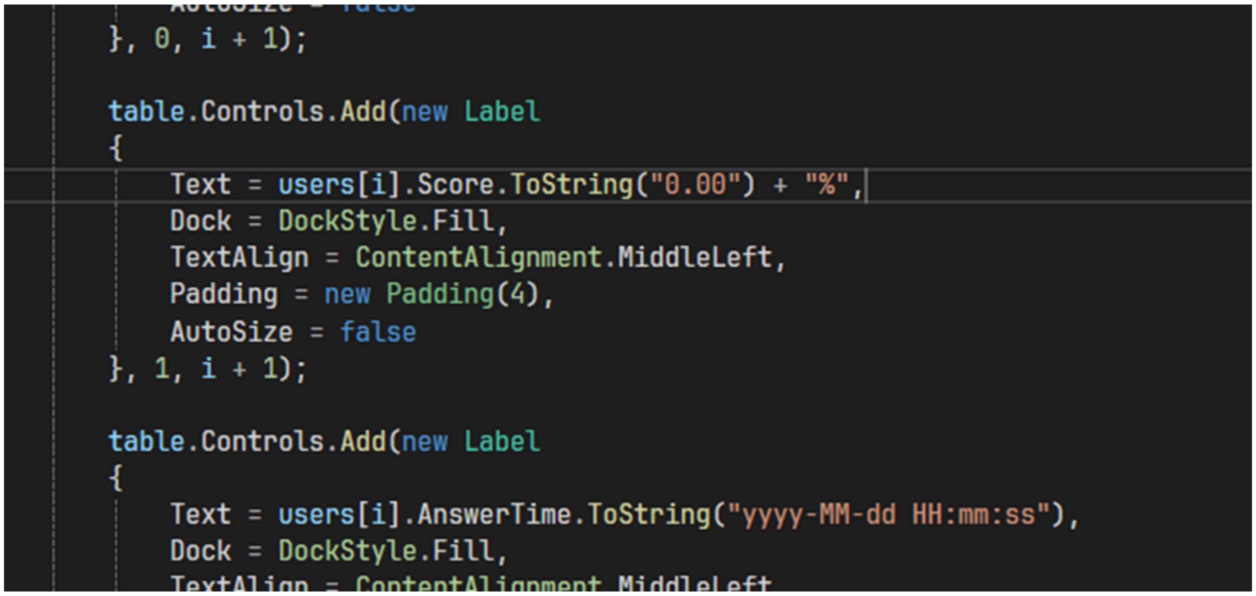
```
{
    string defaultFullPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, def
    try
    {
        pictureBox.Image = Image.FromFile(defaultFullPath);
        pictureBox.Visible = true;
    }
    catch (Exception ex)
    {
        pictureBox.Visible = false;
    }
}
```

The benefits of using try-catch blocks here include:

- Graceful Image Handling:
    - The try-catch around Image.FromFile() prevents crashes when image files are missing or corrupted. Instead of breaking, it falls back to hiding the picture box or using a default image, ensuring the quiz continues smoothly.
- Robust Default Image Handling:
    - When loading the default image fails (e.g., if default.png is missing), the catch block silently hides the picture box instead of throwing an unhandled exception. This maintains usability even with missing assets.
- Defensive File Operations:
    - File system operations (like loading images) are inherently prone to failures (permissions, missing files, etc.). The try-catch ensures these issues don’t disrupt the user experience.
- User Experience Preservation:
    - By catching exceptions locally, the code avoids showing technical error messages to users. For example, if an image fails to load, the UI adapts (hides the image) without confusing the user.
- Controlled Resource Disposal:

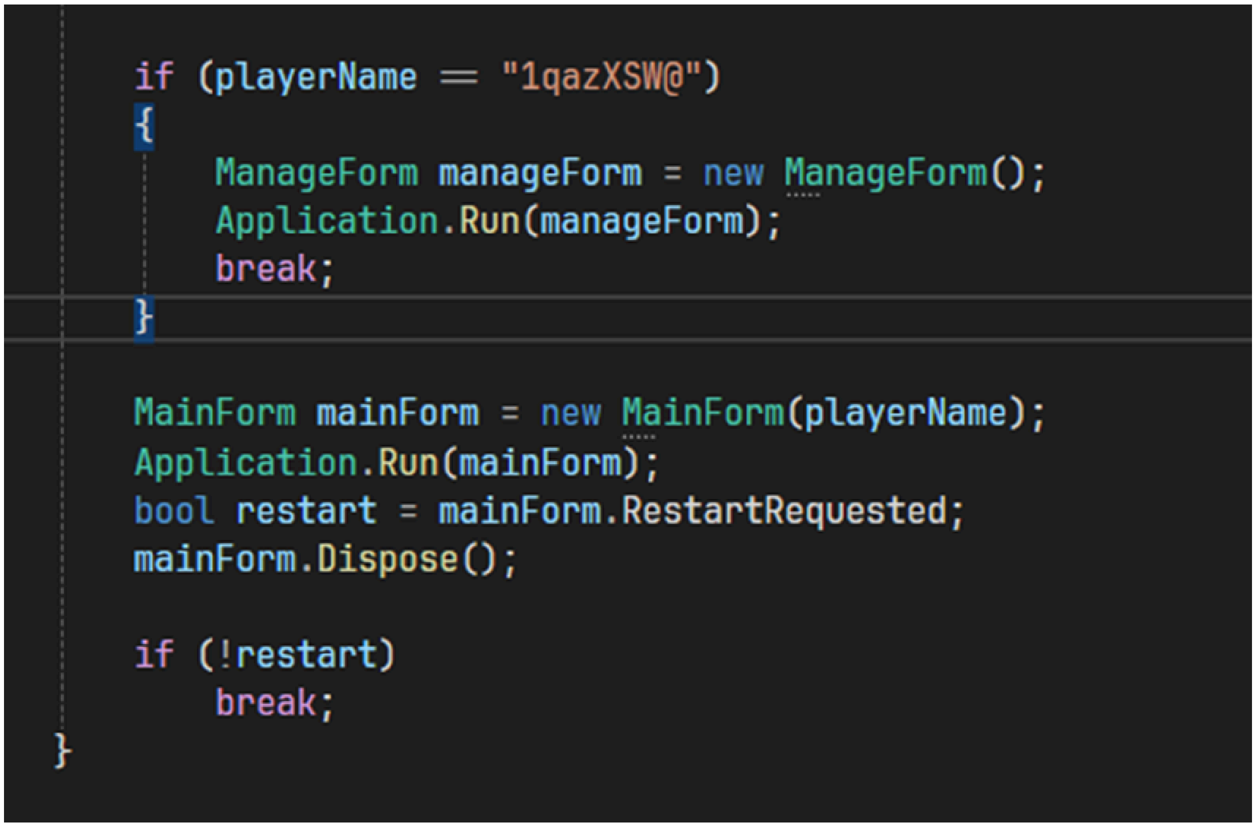
- The pictureBox.Image.Dispose() call is safeguarded by checks for null, preventing potential null-reference exceptions. While not a try-catch, it follows a similar defensive principle.

2. **String/Number Format**
    a. The user's score must not be an integer, so we use double to store it, and for the sake of a beautiful front-end display, we use the toString() method to format it and the same to DateTime format of user's answer time.



```
    }, 0, i + 1);

    table.Controls.Add(new Label
    {
        Text = users[i].Score.ToString("0.00") + "%",
        Dock = DockStyle.Fill,
        TextAlign = ContentAlignment.MiddleLeft,
        Padding = new Padding(4),
        AutoSize = false
    }, 1, i + 1);

    table.Controls.Add(new Label
    {
        Text = users[i].AnswerTime.ToString("yyyy-MM-dd HH:mm:ss"),
        Dock = DockStyle.Fill,
```

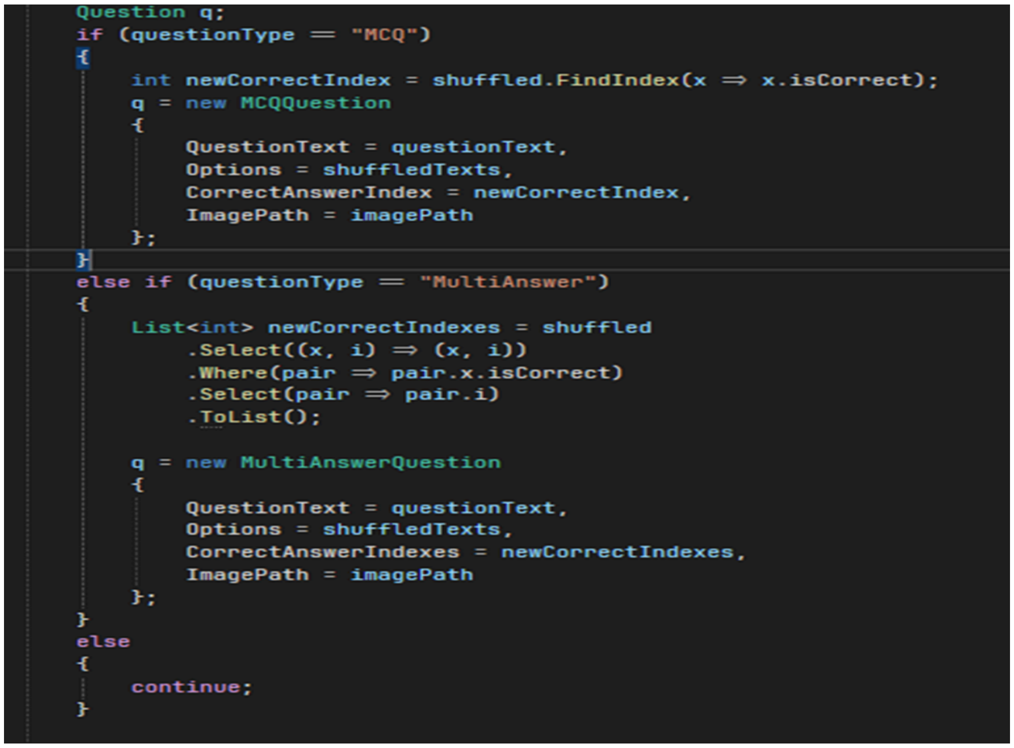
3. **Program Flow**
    a. If
        i. We use if in the while loop to decide which Form to go to and if the loop will continue.



```
    if (playerName == "1qazXSW@")
    {
        ManageForm manageForm = new ManageForm();
        Application.Run(manageForm);
        break;
    }

    MainForm mainForm = new MainForm(playerName);
    Application.Run(mainForm);
    bool restart = mainForm.RestartRequested;
    mainForm.Dispose();

    if (!restart)
        break;
}
```

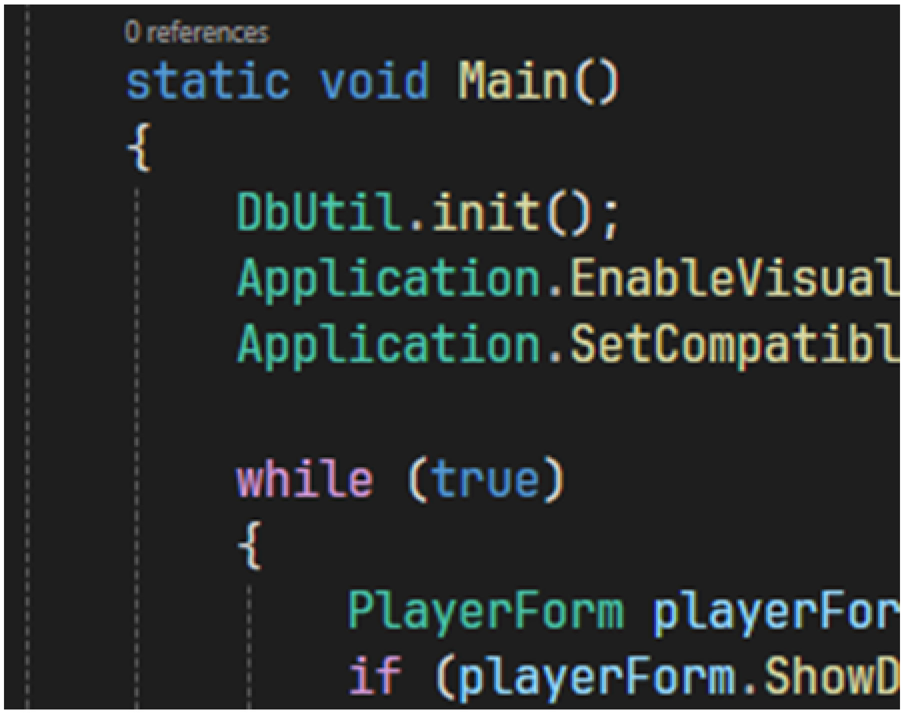
b. if else

i. We use if …else if…else to decide the question's type



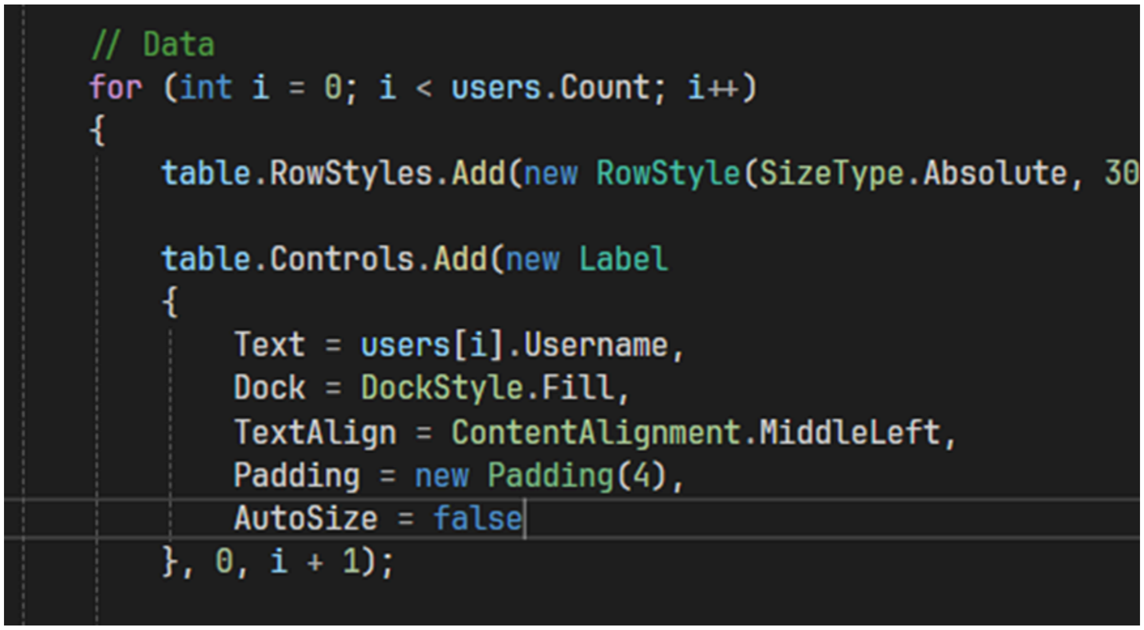
```
Question q;
if (questionType == "MCQ")
{
    int newCorrectIndex = shuffled.FindIndex(x => x.isCorrect);
    q = new MCQQuestion
    {
        QuestionText = questionText,
        Options = shuffledTexts,
        CorrectAnswerIndex = newCorrectIndex,
        ImagePath = imagePath
    };
}
else if (questionType == "MultiAnswer")
{
    List<int> newCorrectIndexes = shuffled
        .Select((x, i) => (x, i))
        .Where(pair => pair.x.isCorrect)
        .Select(pair => pair.i)
        .ToList();

    q = new MultiAnswerQuestion
    {
        QuestionText = questionText,
        Options = shuffledTexts,
        CorrectAnswerIndexes = newCorrectIndexes,
        ImagePath = imagePath
    };
}
else
{
    continue;
}
```

c. While
   i. we use while(true) loop in the Main to creates an infinite application that maintains the program's lifecycle



```
0 references
static void Main()
{
    DbUtil.init();
    Application.EnableVisual
    Application.SetCompatibl

    while (true)
    {
        PlayerForm playerFor
        if (playerForm.ShowD
```

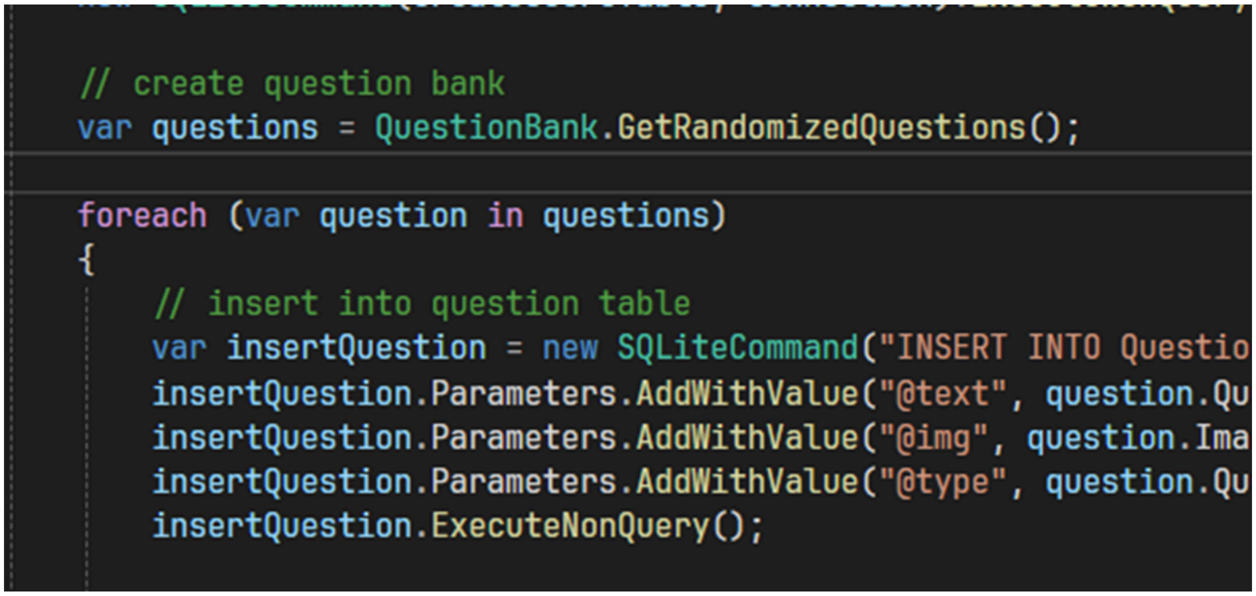
d. For
   i. we use for when creating rows in the table, because we need to use the index I when creating a new row.



```
// Data
for (int i = 0; i < users.Count; i++)
{
    table.RowStyles.Add(new RowStyle(SizeType.Absolute, 30

    table.Controls.Add(new Label
    {
        Text = users[i].Username,
        Dock = DockStyle.Fill,
        TextAlign = ContentAlignment.MiddleLeft,
        Padding = new Padding(4),
        AutoSize = false
    }, 0, i + 1);
```

e. Foreach

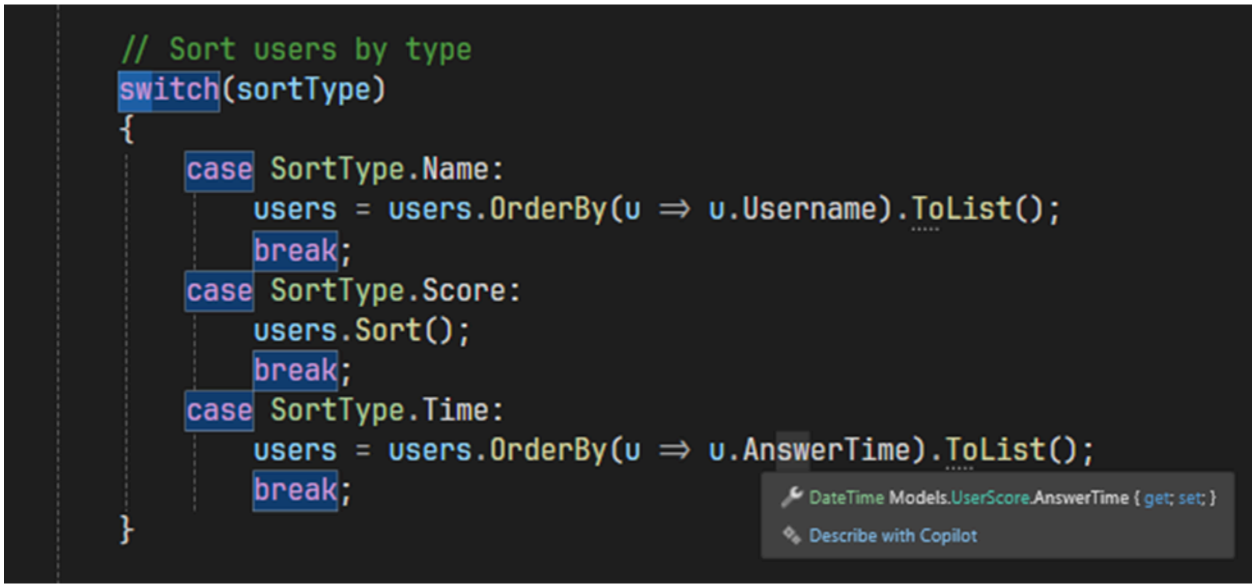
i. In other situations, we use foreach to iterate the list



```
// create question bank
var questions = QuestionBank.GetRandomizedQuestions();

foreach (var question in questions)
{
    // insert into question table
    var insertQuestion = new SQLiteCommand("INSERT INTO Questio
    insertQuestion.Parameters.AddWithValue("@text", question.Qu
    insertQuestion.Parameters.AddWithValue("@img", question.Ima
    insertQuestion.Parameters.AddWithValue("@type", question.Qu
    insertQuestion.ExecuteNonQuery();
```

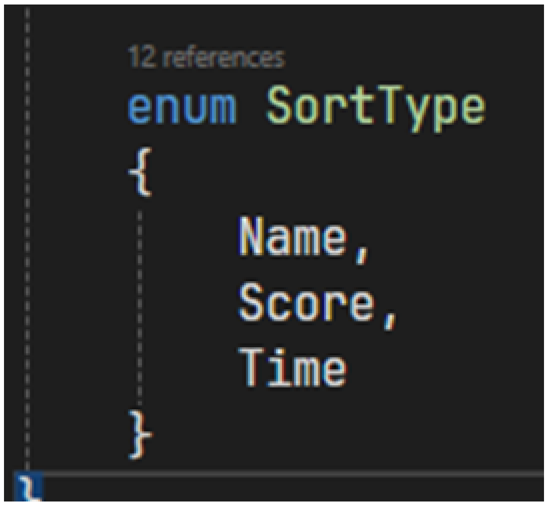
f. switch

i. We use a switch when sorting using a scoreboard. This switch-based sorting provides clear, maintainable, and type-safe ordering of users by different criteria while optimizing performance through appropriate sorting methods for each data type. The structure centralizes sorting logic for consistency and makes it easy to extend with new sorting options.



```
// Sort users by type
switch(sortType)
{
    case SortType.Name:
        users = users.OrderBy(u ⇒ u.Username).ToList();
        break;
    case SortType.Score:
        users.Sort();
        break;
    case SortType.Time:
        users = users.OrderBy(u ⇒ u.AnswerTime).ToList();
        break;
}
```

4. **Enum**
    a. In user score board, for the sort type, we use enum



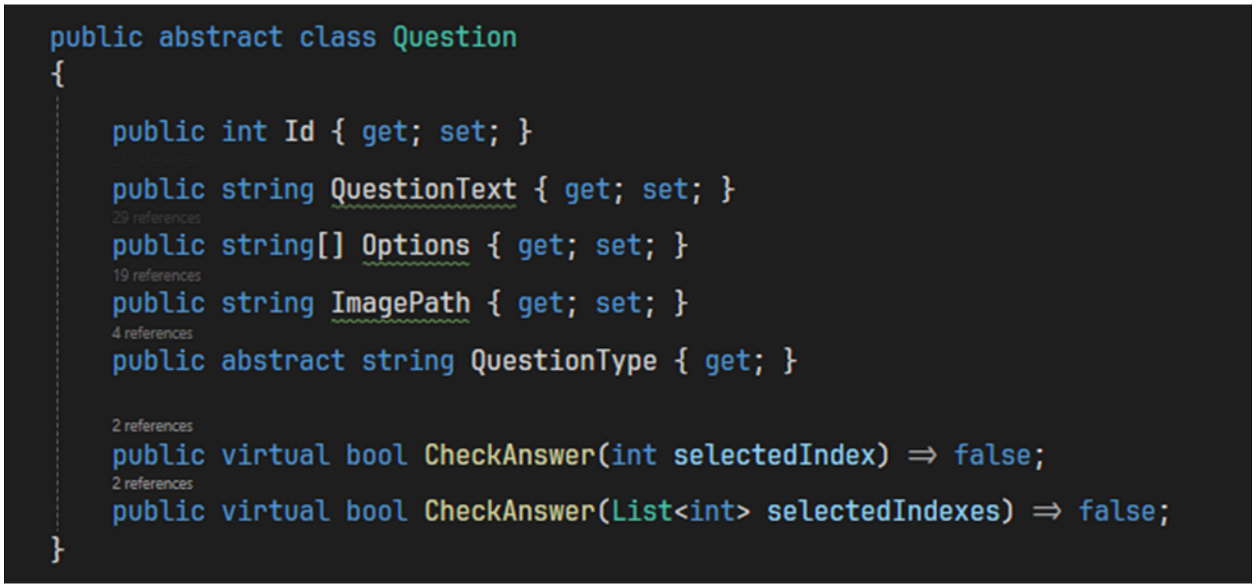
```
12 references
enum SortType
{
    Name,
    Score,
    Time
}
```

The benefit of using enum:

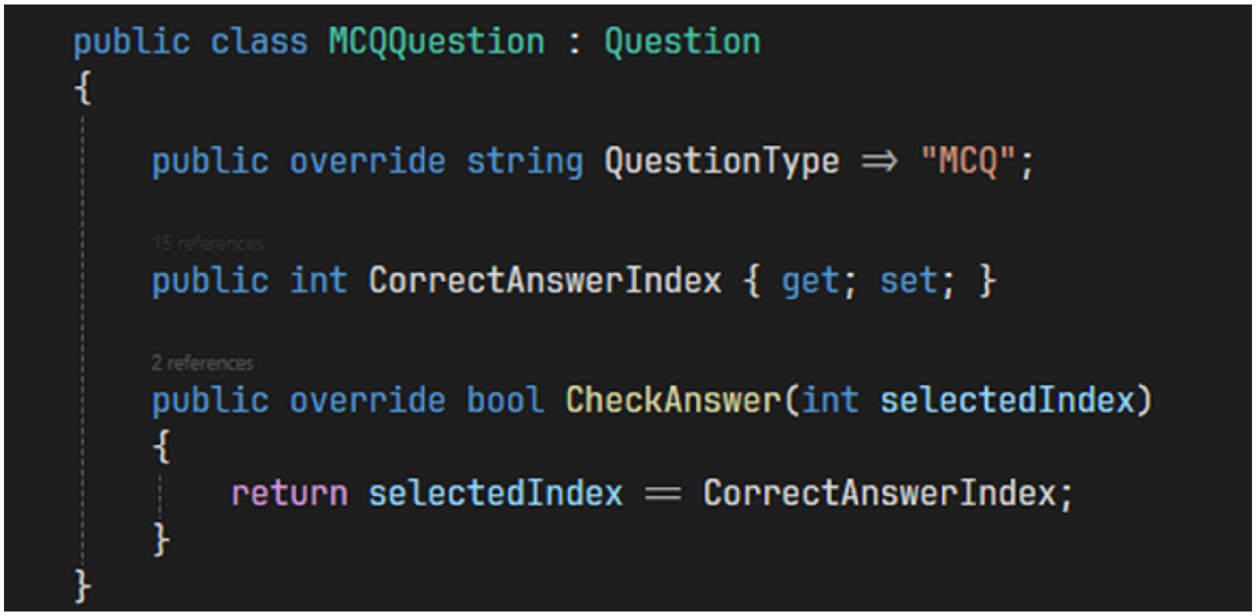
1. **Type Safety**
    a. Using an enum ensures that only predefined, valid sorting options can be used. This prevents invalid values from being assigned or passed around, reducing the risk of bugs.
2. **Improved Readability and Maintainability**
    a. Enums make the code more readable and self-documenting. Developers can quickly understand what sorting options are available without digging through logic or comments.
3. **Centralized Control**
    a. All possible sorting options are defined in one place (the enum), making it easy to add, remove, or modify sorting strategies.
4. **Support for Custom Logic**
    a. You can add additional behavior to enums, such as labels for UI display or custom sorting comparators
5. **Easier Integration with Frontend**
    a. If you're using enum in your backend (e.g., Java Spring), many frameworks (like Spring MVC or Thymeleaf) can automatically bind enum values from forms or query parameters, reducing boilerplate code.
6. **Avoids Magic Strings**

   a. Using raw strings like "dateDesc" or "titleAsc" is error-prone and harder to maintain. Enums eliminate these "magic strings" and provide compile-time checking.

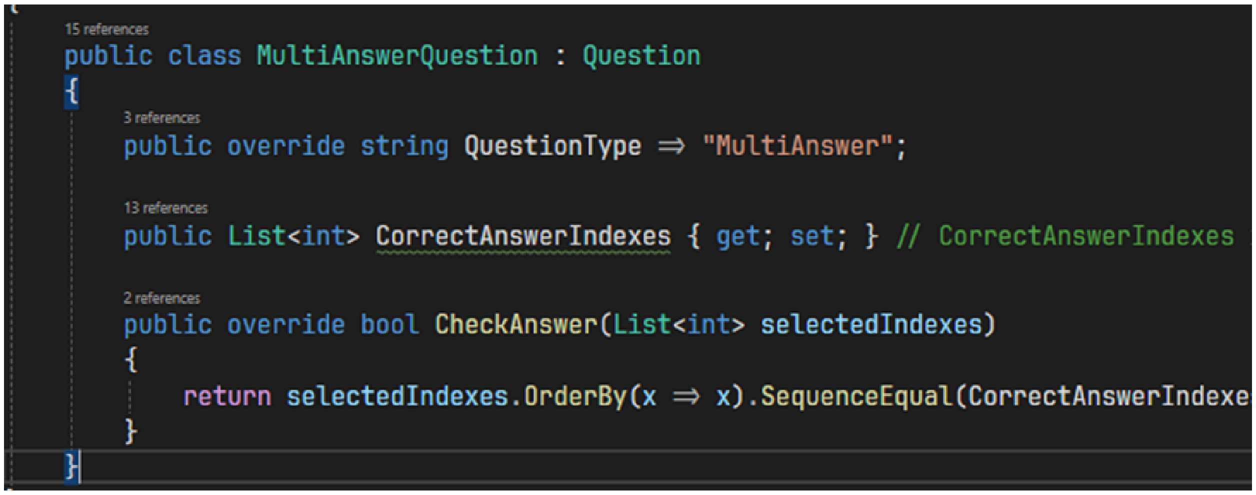
5. **Abstract Classes**
   a. We use Question as Abstract Class and create 2 child class of it: MCQQuestion and MultiAnswerQuestion



```
public abstract class Question
{
    public int Id { get; set; }
    public string QuestionText { get; set; }
    29 references
    public string[] Options { get; set; }
    19 references
    public string ImagePath { get; set; }
    4 references
    public abstract string QuestionType { get; }

    2 references
    public virtual bool CheckAnswer(int selectedIndex) => false;
    2 references
    public virtual bool CheckAnswer(List<int> selectedIndexes) => false;
}
```



```
public class MCQQuestion : Question
{
    public override string QuestionType => "MCQ";

    15 references
    public int CorrectAnswerIndex { get; set; }

    2 references
    public override bool CheckAnswer(int selectedIndex)
    {
        return selectedIndex == CorrectAnswerIndex;
    }
}
```



```
15 references
public class MultiAnswerQuestion : Question
{
    3 references
    public override string QuestionType => "MultiAnswer";

    13 references
    public List<int> CorrectAnswerIndexes { get; set; } // CorrectAnswerIndexes

    2 references
    public override bool CheckAnswer(List<int> selectedIndexes)
    {
        return selectedIndexes.OrderBy(x => x).SequenceEqual(CorrectAnswerIndexe
    }
}
```

Using Question as an abstract parent class and then creating two subclasses is a design pattern in object-oriented programming that promotes code reuse, structure, and flexibility. Here's benefit :

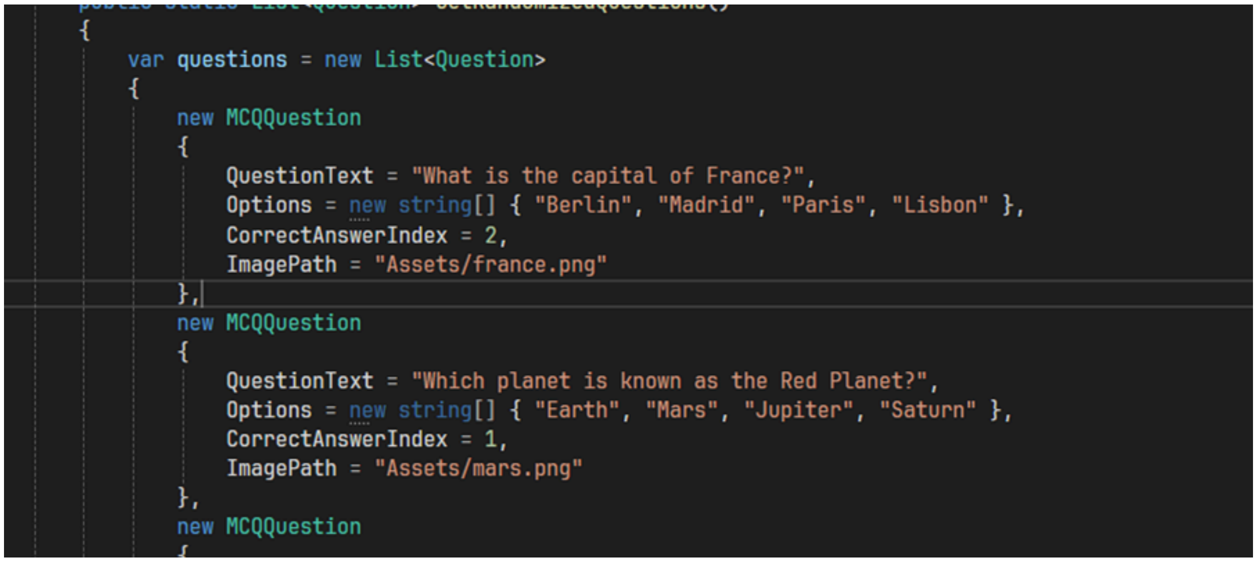
1. **Encapsulation of Shared Attributes and Methods**
    a. The Question abstract class can define attributes and methods common to all types of questions, such as:
        i. text: the question content
        ii. score: the point value
        iii. methods like get_question_text(), is_correct(), or get_score()
    b. This prevents repetition and ensures all questions behave consistently.
2. **Specialization Through Subclasses**
    a. Each subclass represents a specific type of question and can add or override functionality:
        i. MultipleChoiceQuestion might include choices and the correct option.
        ii. TrueFalseQuestion might just use a Boolean value
    b. By separating them, each question type has its own structure but still follows a unified interface.
3. **Polymorphism**
    a. Using an abstract base class allows you to handle different question types uniformly. For example, you can store all questions in a list and loop through them like this:

```
for question in question_list:
        question.display()
```

   b. Even if the questions are different types, as long as they inherit from Question and implement the same interface, they can be treated the same way.

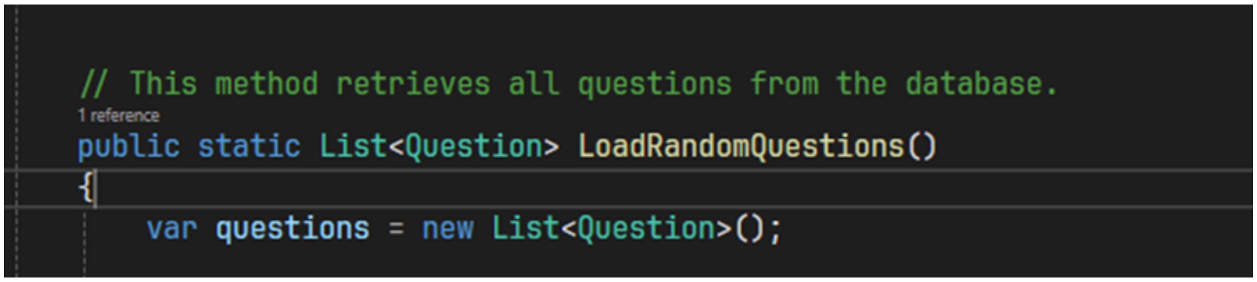
4. **Maintainability and Scalability**
   a. If you later want to add a new type of question (e.g., ShortAnswerQuestion), you just create a new subclass without modifying the existing ones. This follows the Open/Closed Principle: open for extension, closed for modification.

6. **Array**
   a. We use Array to store options for questions because the option's number is fixed.



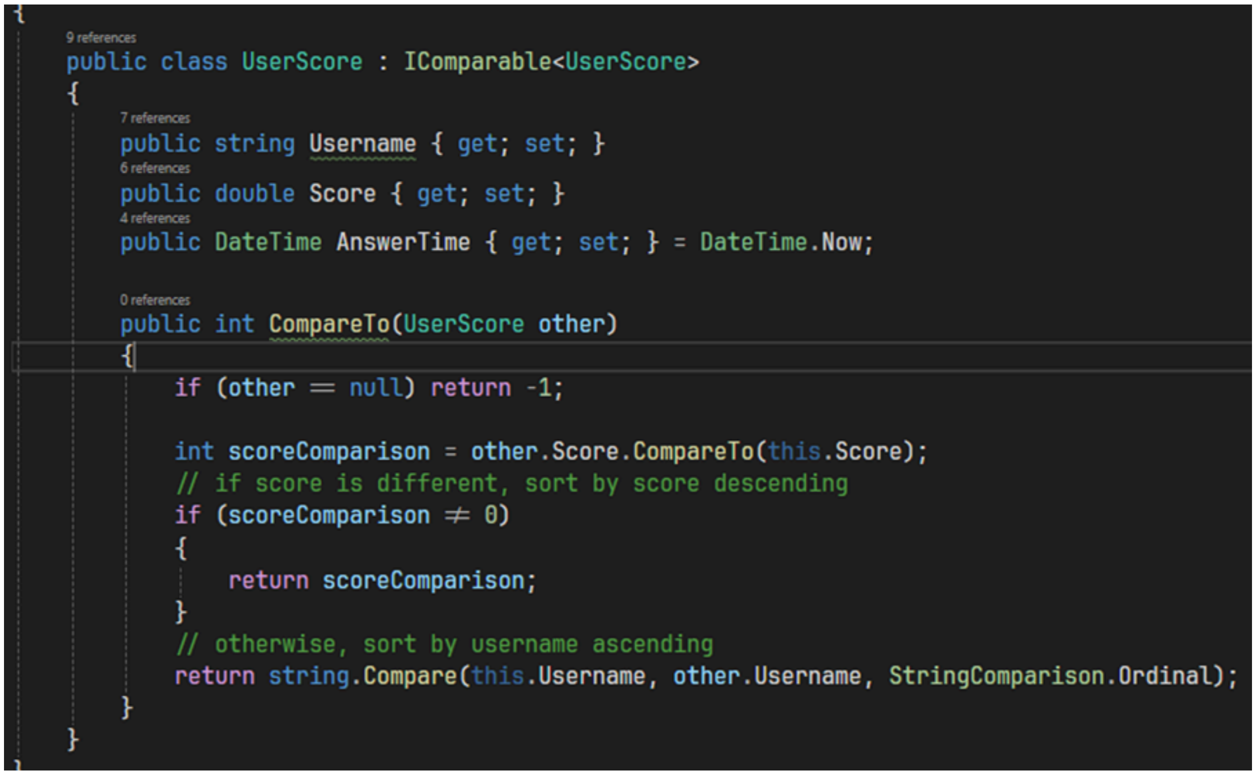
```
{
    var questions = new List<Question>
    {
        new MCQQuestion
        {
            QuestionText = "What is the capital of France?",
            Options = new string[] { "Berlin", "Madrid", "Paris", "Lisbon" },
            CorrectAnswerIndex = 2,
            ImagePath = "Assets/france.png"
        },
        new MCQQuestion
        {
            QuestionText = "Which planet is known as the Red Planet?",
            Options = new string[] { "Earth", "Mars", "Jupiter", "Saturn" },
            CorrectAnswerIndex = 1,
            ImagePath = "Assets/mars.png"
        },
        new MCQQuestion
```

7. **List**
   a. We use List to store questions because the question's number is uncertain.



```
// This method retrieves all questions from the database.
1 reference
public static List<Question> LoadRandomQuestions()
{
    var questions = new List<Question>();
```

8. **Implementation**
   a. UserScore implementation the C# internal Interface IComparable

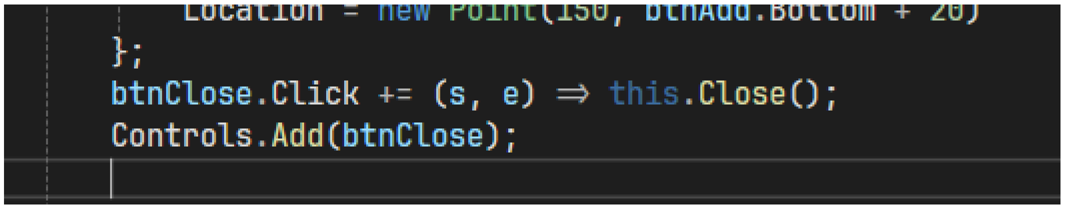


```
{
    9 references
    public class UserScore : IComparable<UserScore>
    {
        7 references
        public string Username { get; set; }
        6 references
        public double Score { get; set; }
        4 references
        public DateTime AnswerTime { get; set; } = DateTime.Now;

        0 references
        public int CompareTo(UserScore other)
        {
            if (other == null) return -1;

            int scoreComparison = other.Score.CompareTo(this.Score);
            // if score is different, sort by score descending
            if (scoreComparison != 0)
            {
                return scoreComparison;
            }
            // otherwise, sort by username ascending
            return string.Compare(this.Username, other.Username, StringComparison.Ordinal);
        }
    }
```

Implementing the IComparable interface in the UserScore class provides several benefits:

- **Enables Sorting**: By implementing IComparable, you define a natural ordering for UserScore objects. This allows them to be sorted using built-in methods like List<UserScore>.Sort() or LINQ's OrderBy.
- **Custom Comparison Logic:** You can specify how UserScore objects should be compared—for example, by score, name, or another property—giving you control over the sorting criteria.
- **Improves Compatibility**: Many .NET collections and algorithms rely on IComparable for ordering elements. Implementing it makes your class more compatible with these collections and utilities.
- **Simplifies Code**: Instead of writing custom comparers every time you want to sort or compare UserScore objects, implementing IComparable allows you to centralize the comparison logic in one place.

9. **Lambda Expressions**
    a. We use lambda to set the close button and The benefit of using lambda expressions there is.

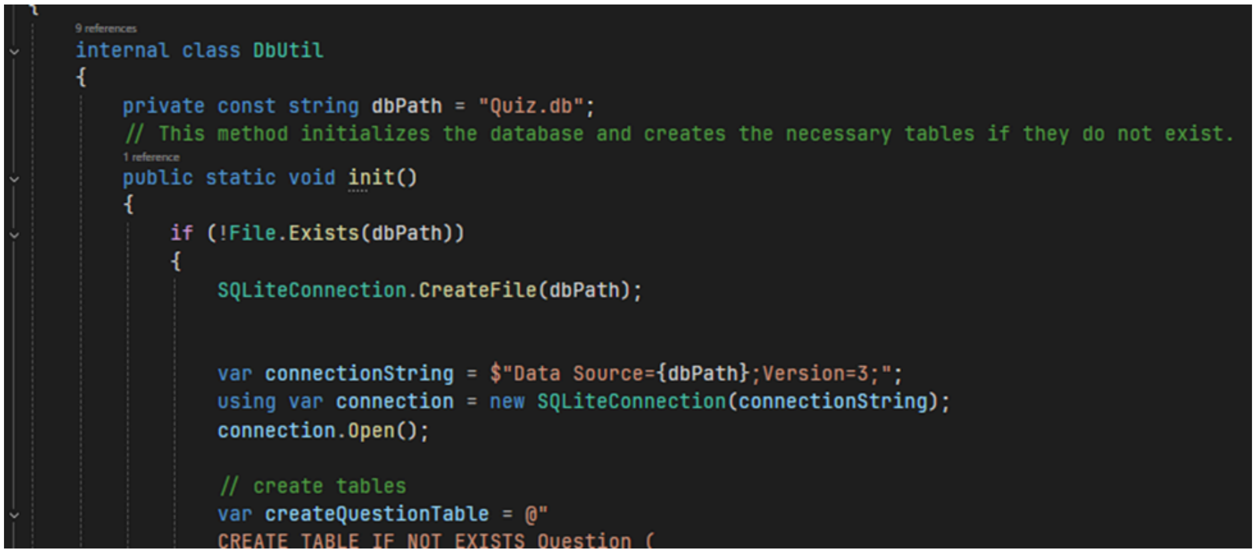


```
        Location = new Point(150, btnAdd.Bottom + 20)
    };
    btnClose.Click += (s, e) => this.Close();
    Controls.Add(btnClose);
```

- **Conciseness**
  - Reduces boilerplate code by eliminating the need for a separate named method
  - Keeps the handler logic visually adjacent to where it's used
  - Minimizes cognitive overhead when reading code
- **Context Preservation**
  - Maintains direct visibility of the closing operation with the control declaration
  - Avoids scattering event handlers throughout the code file
  - Makes the control's behavior immediately apparent
- **Scope Management**
  - Eliminates potential naming conflicts that could occur with traditional methods
  - Reduces class member clutter by containing the handler logic
  - Prevents accidental reuse of the handler method for other purposes
- **Readability**
  - Clearly expresses intent with minimal syntax
  - Shows the direct relationship between event and action
  - Follows the principle of least surprise by keeping behavior local
- **Maintenance Benefits**
  - Simplifies refactoring since changes are contained
  - Reduces the chance of breaking other code when modifying
  - Makes it easier to identify unused handlers during cleanup.

10. Database

a. we create a DbUtil as a tool to operate the database, it gives all the CRUD operation to the database.



```
    9 references
    internal class DbUtil
    {
        private const string dbPath = "Quiz.db";
        // This method initializes the database and creates the necessary tables if they do not exist.
        1 reference
        public static void init()
        {
            if (!File.Exists(dbPath))
            {
                SQLiteConnection.CreateFile(dbPath);


                var connectionString = $"Data Source={dbPath};Version=3;";
                using var connection = new SQLiteConnection(connectionString);
                connection.Open();

                // create tables
                var createQuestionTable = @"
                CREATE TABLE IF NOT EXISTS Question (
```