Problem Statement

Project Title: MiniSocial – Console Based Social Media Application

You are required to design a **console-based mini social media platform** where users can register, log in, create posts, follow other users, and view a personalized timeline.

The system must demonstrate **core C# and OOP concepts** including:

- Namespaces
- Classes and Interfaces
- Access modifiers
- Events and Delegates
- Generics
- Exception handling
- File handling (JSON + logs)
- Extension methods
- LINQ usage

```
MiniSocialMedia.cs

│

└── MiniSocialMedia

    │

    ├── SocialException(class)

    │   ├── SocialException(method)

    │   └── SocialException

    │

    ├── IPostable (interface)

    │   ├── AddPost(method)

    │   └── GetPosts
```

```
│
├── User (partial)
│   ├── Username
│   ├── Email
│   ├── _posts
│   ├── _following
│   ├── User
│   ├── Follow
│   ├── IsFollowing
│   ├── AddPost
│   ├── GetPosts
│   ├── CompareTo
│   ├── ToString
│   └── OnNewPost
│
├── User (partial)
│   └── GetDisplayName
│
├── Post
│   ├── Author
│   ├── Content
│   ├── CreatedAt
│   ├── Post
│   └── ToString
```

```
│
├── Repository
│   ├── _items
│   ├── Add
│   ├── GetAll
│   └── Find
│
├── SocialUtils
│   └── FormatTimeAgo
│
├── Program
│   ├── _users
│   ├── _currentUser
│   ├── _dataFile
│   ├── Main
│   ├── ShowLoginMenu
│   ├── Register
│   ├── Login
│   ├── ShowMainMenu
│   ├── PostMessage
│   ├── ShowTimeline
│   ├── ShowPosts
│   ├── FollowUser
│   ├── ListUsers
│   ├── SaveData
│   ├── LoadData
│   ├── LogError
│   └── ConsoleColorWrite
│
└── UserExtensions
    └── GetFollowingNames
```

Namespace Specification

Namespace Name

MiniSocialMedia

All classes, interfaces, and utilities must exist inside this namespace.

Task 1: Create Custom Exception Handling

Class Name

SocialException

Base Class

Exception

Data Members

None (inherits from Exception)

Constructors

Constructor	Parameters	Purpose
SocialException	message : string	Stores a custom error message
SocialException	message : string, inner : Exception	Stores message with inner exception

What This Class Must Do

- Represent **business-level errors** (invalid email, self-follow, long post, etc.)
- Be thrown instead of generic exceptions where rules are violated

Test Case

Input:
Email = `"abc@"`
Expected Output:
Error message: `"Invalid email format"`

Task 2: Define Posting Capability Using Interface

Interface Name

IPostable

Methods

Method Name	Return Type	Parameters	Purpose
AddPost	void	content : string	Add a new post
GetPosts	IReadOnlyList	none	Return user's posts

Expected Outcome

- Any class implementing `IPostable` must allow posting and viewing posts

Test Case

Input:
content = `"Hello World"`
Expected Output:
Post is added successfully

Task : Create the Post Class

Class Name

`Post`

Purpose

- Represent a single post created by a user in the social media system.
- Store author information, post content, and creation time.
- Provide a formatted textual representation of the post.

Task : Define Data Members (Properties)

Property 1

- Name: `Author`
- Type: `User`
- Access: public
- Assignment rule:
    - Value must be assigned only during object creation.
    - Value must never be changed later.

Property 2

- Name: `Content`
- Type: `string`
- Access: public
- Assignment rule:

    - Value must be assigned only during object creation.
    - Value must never be changed later.

Property 3

- Name: `CreatedAt`
- Type: `DateTime`
- Access: public
- Assignment rule:
    - Automatically store the post creation time.
    - Use the current UTC time as the default value.

Task : Implement the Constructor

Constructor Inputs

- `author` : User
- `content` : string

Constructor Instructions

1. Accept a user object through the parameter named `author`.
2. Check whether `author` is null.
    - If null:
        - Stop object creation.
        - Raise an argument-related exception mentioning `author`.
3. Store the valid user object as the post's author.
4. Store the provided `content` as the post content.
5. Do not perform formatting or trimming inside the constructor.

Input–Output Test Case (Constructor)

Input

- author = valid User object

- content = `"Hello World"`

Expected Outcome

- Post object is created successfully.
- `Author` is set correctly.
- `Content` is stored exactly as provided.
- `CreatedAt` contains the current UTC timestamp.

Task : Implement String Representation of Post

Method Name

`ToString`

Return Type

`string`

Input Parameters

None

Instructions for ToString Method

1. Override the default string representation method.
2. Build a multi-line string containing:
    - Line 1:
        - Author representation
        - A separator symbol
        - Post creation date and time in readable format
    - Line 2:
        - The post content exactly as stored
3. Scan the post content for hashtags using regex
    - A hashtag:
        - Starts with `#`
        - Contains alphabetic characters
4. If one or more hashtags are found:
    - Add a new line starting with the label `Tags:`

    - Display all detected hashtags separated by commas.
    - HINT: sb.AppendJoin(", ", hashtags.Cast<Match>().Select(m => m.Value));
5. Ensure the final returned string has no trailing empty lines.

Input–Output Test Case (Post Display with Tags)

Input

- Author = `@Aman`
- Content = `"Learning C# is fun #dotnet #coding"`

Expected Output

@Aman • Mar 10 14:30

Learning C# is fun #dotnet #coding

Tags: #dotnet, #coding

Input–Output Test Case (Post Display without Tags)

Input

- Author = `@Neha`
- Content = `"Good morning everyone"`

Expected Output

@Neha • Mar 10 09:15

Good morning everyone

Task 3: Create User Entity (Core Class)

Class Name

User : It's a partial class.

Implements

IPostable, IComparable<User>

Properties

Name	Type	Access Modifier
Username	string	Init property
Email	string	init
_posts	List	private
_following	HashSet	private
OnNewPost	Action	public event

HINT: private readonly HashSet<string> _following = new(StringComparer.OrdinalIgnoreCase);

Create Constructor

Step 1: Accept Required Inputs

- Accept two input values:
    - Variable name: `username`

    - Variable name: `email`

Step 2: Validate Username

- Check whether `username` is:
    - null
    - empty
    - only whitespace
- If invalid:
    - Stop user creation.
    - Raise an argument-related exception mentioning `username`.

Step 3: Validate Email Format

- Validate the value of `email` using a pattern-based check.
- The pattern must ensure:
    - Characters exist before `@`
    - Characters exist after `@`
    - A domain extension exists

Step 4: Handle Invalid Email

- If the email does not match the required format:

        - Stop user creation.
        - Raise a custom exception with the message:
            - `"Invalid email format"`

Step 5: Normalize Username

- Remove leading and trailing spaces from `username`.
- Store the cleaned value in the user object.

Step 6: Normalize Email

- Remove leading and trailing spaces from `email`.
- Convert `email` to lowercase.
- Store the final value in the user object.

Input–Output Test Case (User Creation)

Input Values

- `username` = `"  Aman  "`
- `email` = `"Aman@Mail.COM"`

Expected Outcome

- User is created successfully.
- Stored username = `"Aman"`

- Stored email = `"aman@mail.com"`

FOLLOW METHOD

Step 1: Accept Target Username

- Accept one input value:
    - Variable name: `username`
- This represents the user to be followed.

Step 2: Prevent Self-Follow : Hint: StringComparison.OrdinalIgnoreCase

- Compare the input `username` with the current user's stored username.
- Comparison must ignore letter case.
- If both values match:
    - Stop the operation.
    - Raise a custom exception with the message:
        - `"Cannot follow yourself"`

Step 3: Add to Following List

- If validation passes:

    - Add the input `username` to the following collection.
- The collection must:
    - Reject duplicate usernames automatically.
    - Treat uppercase and lowercase usernames as the same.

Input–Output Test Case (Follow)

Input Values

- Current user username = `"Rahul"`
- Follow username = `"rahul"`

Expected Outcome

- Operation fails.
- Error message: `"Cannot follow yourself"`

Input–Output Test Case (Valid Follow)

Input Values

- Current user username = `"Rahul"`
- Follow username = `"Aman"`

Expected Outcome

- Follow operation succeeds.
- `"Aman"` appears once in the following list.

IsFollowing (Lambda Function)

Step 1: Accept Input

- Accept one input value:
    - Variable name: `username`
- This value represents the user whose follow status must be checked.

Step 2: Check Follow Collection

- Check whether the internal following collection contains the input `username`.
- The check must:
    - Ignore differences in letter casing.
    - Use the existing following collection only.

Step 3: Return Result

- Return:
    - `true` if the user is already being followed.
    - `false` if the user is not being followed.

Input–Output Test Case (Follow Check)

Input

- Following list contains: `"Aman", "Neha"`
- `username` = `"aman"`

Expected Output

- `true`

Input–Output Test Case (Not Following)

Input

- Following list contains: `"Aman", "Neha"`
- `username` = `"Rahul"`

Expected Output

- `false`

TASK: Implement Post Notification Event

Event Name

`OnNewPost`

Nullable

Yes

Instructions

- Declare an event named **OnNewPost**.
- The event must use an Action delegate that:
    - Accepts a `Post` object
    - Returns nothing
- The event must be triggered **after** a post is successfully created.
- If no subscriber exists:
    - The event must not cause any runtime error.

Expected Outcome

- External components can subscribe to receive notifications when a new post is created.

AddPost method: return type : void

Step 1: Accept Post Content

- Accept one input value:
    - Variable name: `content`
- This value represents the text of the post.

Step 2: Validate Post Content Presence

- Check whether `content` is:
    - null
    - empty
    - contains only whitespace
- If invalid:
    - Stop the operation.
    - Raise an argument-related exception with the message:
        - `"Post content cannot be empty"`

Step 3: Validate Post Length

- Check the length of `content`.
- If the number of characters exceeds `280`:
    - Stop the operation.
    - Raise a custom exception with the message:
        - `"Post too long (max 280 characters)"`

Step 4: Normalize Post Content

- Remove leading and trailing spaces from `content`.
- Use the cleaned value for post creation.

Step 5: Create Post Object

- Create a new post using:
    - The current user as the author.
    - The cleaned post content.

Step 6: Store the Post

- Add the newly created post to the user's internal post collection.
- Do not expose the collection directly.

Step 7: Trigger Post Notification

- If a post notification event exists:
    - Trigger it.
    - Pass the newly created post as the event data.

Input–Output Test Case (Valid Post)

Input

- `content` = `"Learning C# is fun!"`

Expected Outcome

- Post is created successfully.
- Post is added to the user's post list.

- Notification event is triggered.

Input–Output Test Case (Empty Post)

Input

- `content` = `"  "`

Expected Outcome

- Operation fails.
- Error message:
  - `"Post content cannot be empty"`

Input–Output Test Case (Long Post)

Input

- `content` length = `300`

Expected Outcome

- Operation fails.
- Error message:
  - `"Post too long (max 280 characters)"`

TASK:

Method Name

`GetPosts`

Return Type

`IReadOnlyList<Post>`

Input Parameters

None

Instructions

- Create a method named **GetPosts**.
- The method must return the user's post collection in **read-only form**.
- Do not allow external code to:
    - Add posts
    - Remove posts
    - Modify existing posts
- The method must only expose a safe view of the internal post list.

Expected Outcome

- External callers can read posts but cannot change them.

Input–Output Test Case

Input

- Internal post list contains 3 posts.

Output

- A read-only list containing the same 3 posts.
- Any attempt to modify the returned list must fail.

Task : Implement User Comparison Logic

Method Name

`CompareTo`

Return Type

`int`

Input Parameters

- `other` : User (nullable)

Instructions

- Create a comparison method named **CompareTo**.
- First, check whether `other` is null.
    - If `other` is null, return a positive value.
- If `other` is not null:
    - Compare the current user’s `Username` with `other.Username`.
    - HINT: StringComparison.OrdinalIgnoreCase
    - Perform comparison without considering letter case.
- Return:
    - A negative value if current user comes before `other`

    - Zero if both usernames are equal
    - A positive value if current user comes after `other`

Expected Outcome

- Users can be sorted alphabetically by username.

Input–Output Test Case

Input

- Current user username: `"Aman"`
- Other user username: `"Neha"`

Output

- Negative integer value (Aman comes before Neha)

Task : Implement String Representation

Method Name

`ToString`

Return Type

`string`

Input Parameters

None

Instructions

- Override the default string representation method.
- Return a string that:
    - Starts with `@`
    - Followed by the user's username
- Do not include email or any other data.

Expected Outcome

- User is displayed in a social-media-style format.

Input–Output Test Case

Input

- Username = `"Rahul"`

Output

- `"@Rahul"`

TASK:

Class Name

`User` (partial class)

Method Name

`GetDisplayName`

Return Type

`string`

Input Parameters

None

Instructions

1. Extend the existing **User** class using a partial class definition.
2. Implement a method named **GetDisplayName**.
3. The method must return a single formatted string.
4. The returned string must include:
    - The literal text: `User:`
    - The value of the user’s `Username`
    - The value of the user’s `Email`

The format of the returned string must be exactly:

```
User: <Username> <Email>
```

5. where:

    - `<Username>` is replaced with the stored username
    - `<Email>` is enclosed inside angle brackets

Expected Outcome

- The method provides a readable identity string for the user.
- The method must not modify any user data.
- The method must not accept any input parameters.

Input–Output Test Case

Input Values

- Username = `"Aman"`
- Email = `"aman@mail.com"`

Expected Output

```
User: Aman <aman@mail.com>
```

Task : Create a Generic Repository Class with Constraints

Class Name

`Repository<T>`

Generic Constraint

- Restrict the generic type parameter `T` to **reference types only**.

Instruction

- Ensure the repository can store only class-type objects.
- Do not allow value types to be used with this repository.

Task : Define Internal Storage

Data Member

- Name: `_items`
- Type: List of `T`
- Access Level: private
- Mutability Rule:
  - The reference must not be reassigned after initialization.
- Initialization Rule:
  - Initialize automatically when the repository object is created.

Instruction

- This collection is the **only place** where items are stored.
- Do not expose this collection directly outside the class.

Task : Implement Add Operation

Method Name

`Add (lambda method)`

Return Type

`void`

Input Parameter

- `item` : T

Instructions

1. Accept one object of type `T`.
2. Add the object to the internal storage collection.
3. Do not return any value.
4. Do not perform null checks or validation inside this method.

Input–Output Test Case (Add)

Input

- `item` = valid object of type `T`

Expected Outcome

- Object is stored in `_items`.
- Total stored item count increases by one.

Task : Implement Retrieve-All Operation

Method Name

`GetAll (lambda method)`

Return Type

`IReadOnlyList<T>`

Input Parameters

None

Instructions

1. Return all stored items as a **read-only collection**.
2. Prevent external callers from modifying the stored data.
3. Maintain the original insertion order.

Input–Output Test Case (GetAll)

Input

- Repository contains 2 stored objects.

Expected Output

- Read-only list containing those same 2 objects.
- Any attempt to add or remove items from the returned list must fail.

Task : Implement Search Operation

Method Name

`Find (Lambda Method)`

Return Type

`T` (nullable)

Input Parameter

- `match` : Predicate<T>

Instructions

1. Accept a predicate that defines a search condition.
2. Search the internal collection using the predicate.
3. Return:
    - The first matching object if found.
    - Null if no matching object exists.
4. Stop searching once a match is found.

Input–Output Test Case (Find – Found)

Input

- Stored items: User objects with usernames `"Aman"`, `"Neha"`
- Predicate: username equals `"Neha"`

Expected Output

- User object with username `"Neha"`

Input–Output Test Case (Find – Not Found)

Input

- Stored items: User objects with usernames `"Aman"`, `"Neha"`
- Predicate: username equals `"Rahul"`

Expected Output

- Null value

Task: Implement Time-Ago Formatting Utility

Class Name

`SocialUtils`

Class Type

Static utility class

Method Name

`Static method called FormatTimeAgo`

Method Type

Extension method for date-time values

Return Type

`string`

Input

- A date-time value representing a past time

Instructions

1. Create a static utility class named **SocialUtils**.
2. Inside this class, implement a static method named **FormatTimeAgo**.
3. The method must operate as an extension on date-time values.
4. Calculate the time difference between:
    - The current UTC time
    - The provided date-time value
    - Store it in a variable
5. Return a human-readable string based on the following rules:
    - If the difference is less than **1 minute**, return:
        - `"just now"`
    - If the difference is less than **60 minutes**, return:
        - Number of minutes followed by `" min ago"`
    - If the difference is less than **24 hours**, return:
        - Number of hours followed by `" h ago"`
    - If the difference is **24 hours or more**, return:
        - The date formatted as month and day

Input–Output Test Cases

Test Case 1

Input

- Date-time = current time minus 30 seconds

Expected Output

just now

Test Case 2

Input

- Date-time = current time minus 10 minutes

Expected Output

10 min ago

Test Case 3

Input

- Date-time = current time minus 5 hours

Expected Output

5 h ago

Test Case 4

Input

- Date-time = current time minus 3 days

Expected Output

Mar 12

Instructions – Application Control & Flow (Program Class)

Class Name

`Program`

Static Data Members

`_users`

- Stores all registered users
- Uses a generic repository
- Must persist across the application lifetime

`_currentUser`

- Stores the currently logged-in user
- Must be null when no user is logged in

`_dataFile`

- Stores the file name used for saving application data (social-data.json)

- Must be readonly
- Must be used for both saving and loading

Method Instructions (Top to Bottom)

`Main`

STEP: Application Startup Responsibility

What the Program Must Do

- Act as the **single entry point** of the console application.
- Control the **entire application lifecycle**.
- Maintain global state such as users, login status, and data persistence.

Variables Used (Conceptual Instructions)

- Maintain a **central user repository** to store all registered users.
- Maintain a **current user reference** to track login state.
- Maintain a **single file name** that represents persistent storage.

Expected Outcome

- User data is shared across the entire program.
- Only one user can be logged in at a time.
- All data loads from and saves to the same file.

STEP : Application Title and Header Setup

What to Do

- Set the console window title to identify the application.
- Display a visible application header when the program starts.

Expected Output

```
MiniSocial - Console Edition

=== MiniSocial ===
```

STEP : Load Persistent Data

Method to Call (defined later, called here)

- **Method Name:** `LoadData`
- **Return Type:** `void`
- **Parameters:** none

What the Method Will Do

- Read user data from the file specified by the data file variable.
- Populate the user repository with existing users.
- Handle missing or empty file cases safely.

STEP : Start Continuous Application Loop

What to Do

- Run the application inside an **infinite loop**.
- Ensure the program does not terminate unless manually closed.

Expected Outcome

- Application keeps running until the user exits explicitly.

STEP 6: Check Login State on Each Loop Iteration

Decision Rule

- If **no user is logged in**, show the login menu.
- If **a user is logged in**, show the main application menu.

Method Names Used

- `ShowLoginMenu`
- `ShowMainMenu`

Expected Behavior

Current User State	Action Taken
No active user	Login menu displayed
User logged in	Main menu displayed

STEP 7: Login Menu Responsibilities

Method Used

- **Method Name:** `ShowLoginMenu`
- **Return Type:** `void`
- **Parameters:** none

What This Menu Must Do

- Allow user to register or log in.
- Set the current user only after successful authentication.

Expected Outcome

- On successful login, current user is assigned.
- Application moves to main menu in next loop iteration.

STEP 8: Main Menu Responsibilities

Method Used

- **Method Name:** `ShowMainMenu`
- **Return Type:** `void`
- **Parameters:** none

What This Menu Must Do

- Allow access to application features.
- Provide logout functionality.
- Clear the current user on logout.

Expected Outcome

- User remains logged in until logout.
- Logout returns user to login menu.

STEP 9: Handle Application-Specific Errors

Error Type

- **Exception Name:** `SocialException`

What to Do

- Catch this exception separately.
- Display the error message clearly in a different color.
- If an inner error exists, display it as additional context.

Expected Output Example

Error: Invalid password

→ Password does not match stored credentials

STEP 10: Handle Unexpected Errors

Error Type

- Any general runtime exception.

What to Do

- Display a generic error message.
- Print full exception details for debugging.
- Log the error using a logging method.

Method Used

- **Method Name:** `LogError`
- **Return Type:** `void`
- **Parameters:** exception object

Expected Outcome

- Application does not crash.
- Error details are recorded for later review.

STEP 11: Pause and Reset Between Iterations

What to Do

- Pause execution until the user presses a key.
- Clear the console screen before restarting the loop.

Expected Outcome

- Clean screen for next interaction.
- Smooth user experience between actions.

ShowLoginMenu

Responsibilities

1. Display options:
    - Register
    - Login
    - Exit
2. Read user choice.
3. Based on choice:
    - Call registration method
    - Call login method
    - Save data and exit
4. Handle invalid menu choices.

Register

Responsibilities

1. Ask for username input.
2. Ask for email input.
3. Validate that both inputs are provided.
4. Check that the username does not already exist.
5. Create a new user.
6. Add the user to the repository.
7. Display a welcome message.

Login

Responsibilities

1. Ask for username input.
2. Search for the user in the repository.
3. If user does not exist:
    - Display an error message.
4. If user exists:

    - Set the current user.
    - Display login confirmation.
5. Subscribe to the post-notification event.
6. Display a notification when a followed user posts.

`ShowMainMenu`

Responsibilities

1. Display logged-in user information.
2. Display options:
    - Post message
    - View own posts
    - View timeline
    - Follow user
    - List users
    - Logout
    - Exit and save
3. Read user choice.
4. Call the corresponding method.
5. Handle invalid menu choices.

`PostMessage`

Responsibilities

1. Ensure a user is logged in.
2. Ask for post content.
3. Allow cancellation if input is empty.
4. Create a new post for the current user.
5. Display confirmation message.

`ShowTimeline`

Responsibilities

1. Ensure a user is logged in.
2. Create a timeline list.
3. Add current user’s posts.
4. Add posts from followed users.
5. Sort posts by most recent first.
6. Display the timeline.

`ShowPosts`

Responsibilities

1. Display posts passed to the method.
2. Limit the number of displayed posts.
3. Show formatted post content.
4. Show relative time for each post.
5. Display a separator between posts.
6. Display a message if no posts exist.

`FollowUser`

Responsibilities

1. Ensure a user is logged in.
2. Ask for the username to follow.
3. Allow cancellation if input is empty.
4. Prevent self-following.
5. Check if the target user exists.
6. Add the user to the following list.
7. Display confirmation message.

`ListUsers`

Responsibilities

1. Display all registered users.
2. Sort users alphabetically.

3. Display username and email.

`SaveData`

Responsibilities

1. Collect all users and their data.
2. Serialize user data to JSON format.
3. Save the data to the specified file.
4. Handle and log any errors during saving.

`LoadData`

Responsibilities

1. Check if the data file exists.
2. Read the file contents.
3. Attempt to deserialize saved data.
4. Display load status.
5. Handle and log any errors during loading.

`LogError`

Responsibilities

1. Write error details to a log file.
2. Include:
    - Timestamp
    - Error message
    - Stack trace
3. Ensure logging failure does not crash the app.

`ConsoleColorWrite`

Responsibilities

1. Temporarily change console text color.
2. Display a message.
3. Restore the original console color.

Student Instructions – Helper Extensions

Class Name

`UserExtensions`

Purpose

- Provide helper methods for the User class
- Keep auxiliary logic separate from core entity logic

Method: `GetFollowingNames`

Responsibilities

1. Return a list of usernames that the user is following.
2. Act as a helper for timeline generation.
3. Do not modify user data.
4. Currently acts as a placeholder for future extension.
