Lab 11 Report

Introduction

This lab was all about getting practice with debugging C# code, specifically focusing on console applications. The main tool I used for this was the Visual Studio Debugger, which is built into the IDE. My primary goal was to get comfortable with the process of finding bugs in code that someone else wrote (or even bugs I might introduce myself), figuring out *why* the code wasn't behaving correctly by tracing its execution, and then applying fixes to make it work as intended.

To do this, I worked with two sample C# console games: Snake and Tetris. These were provided as examples of existing codebases where I could look for potential problems. The process involved several key debugging techniques that are common in software development:

- **Setting Breakpoints:** This is where I tell the debugger to pause the program's execution at a specific line of code so I can examine the state of things (like variable values) at that exact moment.
- **Stepping Through Code:** Once paused, I could execute the code one line at a time. There are different ways to step, and I used these frequently:
  - *Step Into (F11):* This runs the current line. If the line calls a method I wrote or have source code for, the debugger jumps *inside* that method and stops at its first line. I used this when I needed to see exactly what was happening inside a particular function.
  - *Step Over (F10):* This also runs the current line, but if it's a method call, the debugger executes the *entire* method in one go (without showing the steps inside), then pauses at the *next* line in the current method. This was useful when I was confident a method worked okay and just wanted to see what happened after it finished.
  - *Step Out (Shift+F11):* If I had stepped into a method, using Step Out would run the rest of that method and pause right after returning to the place

where the method was originally called. It helped me get back to the previous context faster after looking inside a function.

- **Inspecting Variables:** While the program was paused, I could hover over variables or use the Locals/Watch windows in Visual Studio to see their current values. This was essential for understanding if the program's state matched what I expected.

The lab also mentioned the possibility of "mutation," which basically means I could deliberately add small bugs to the code myself if the original games didn't have enough obvious problems to fix. I actually did this for both games to give myself a few more concrete issues to track down using the debugger.

So, my overall workflow for this lab was: select the games (Snake and Tetris), play them and read the code to find initial issues, use the debugger with breakpoints and stepping to understand the program flow and locate the root cause of the bugs (both the original ones and the ones I added), modify the C# code to fix the identified problems, and finally, run the games again to verify that my fixes worked correctly. I used Visual Studio 2022 and the standard .NET SDK (which includes the C# compiler) for everything.

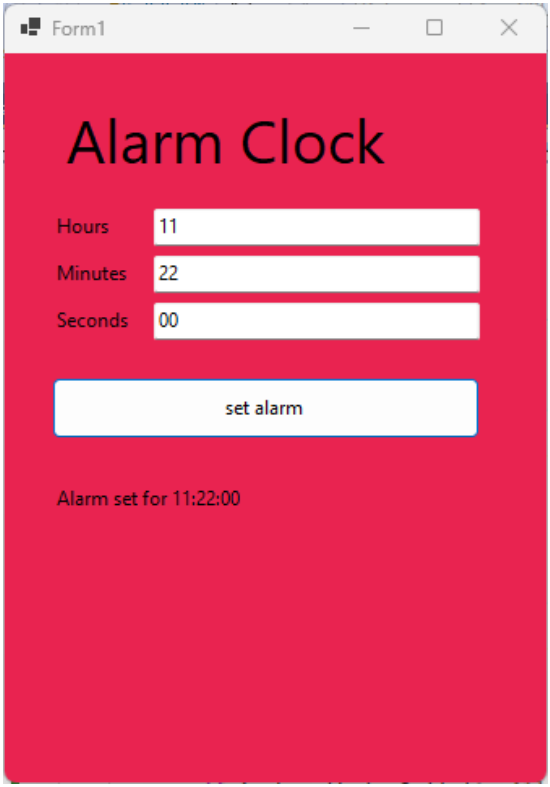
Methodology and Execution

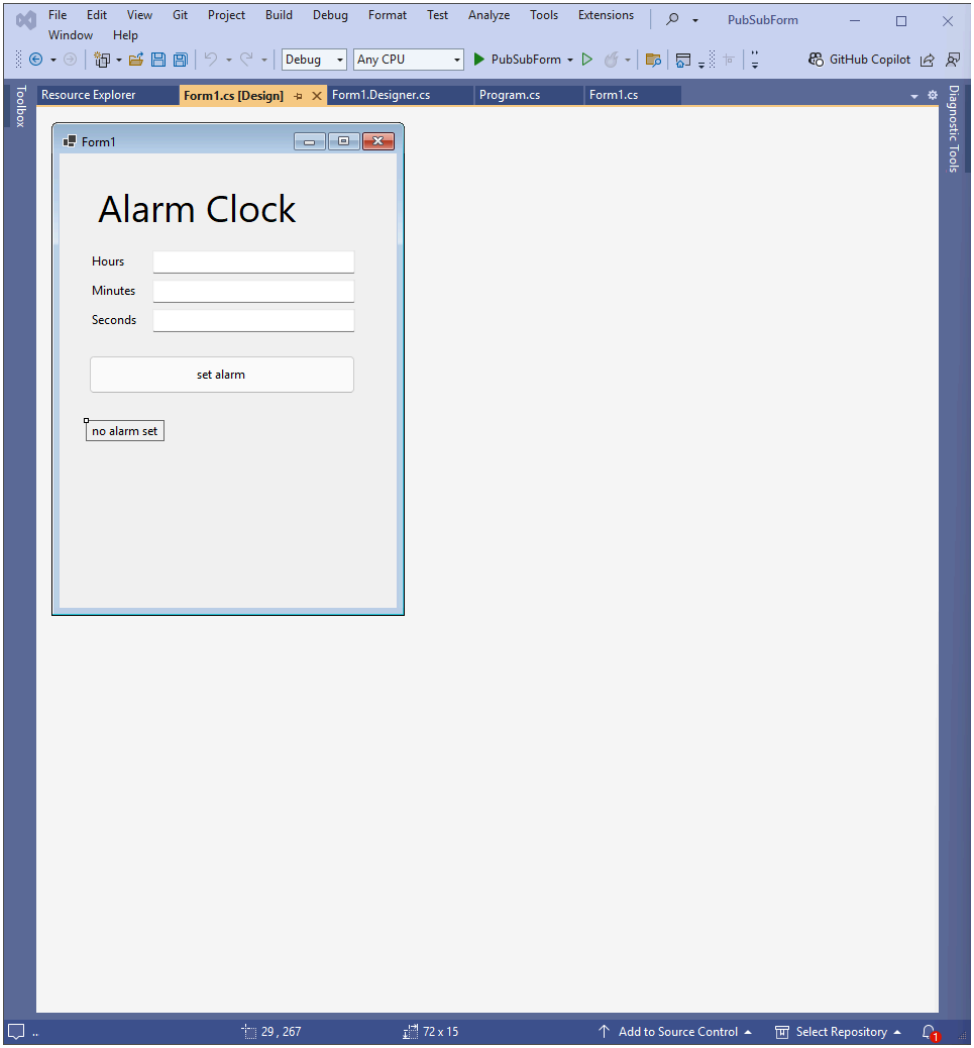
1. Game Selection and Initial Analysis

I started by getting the Snake and Tetris projects from the `dotnet/dotnet-console-games` GitHub repository.

Before trying to debug specific issues, I first ran both games to see how they played and looked through their C# source code in Visual Studio. This initial check revealed several problems:

- **Snake Game:**
    i. When asked to select the speed, pressing the Escape key didn't quit or go back like I expected it might.
    ii. Visually, the snake seemed to move faster vertically (up/down) than horizontally (left/right).
    iii. If I resized the console window while playing, the game would just disappear (crash) without any error message.
    iv. After the snake crashed into itself or a wall, the game ended, but it didn't show the final score.
- **Tetris Game:**
    i. If the game started in a small console window, it showed a message asking to resize. If I pressed Enter *before* actually resizing, the game would just hang.
    ii. There was a specific sequence: resize the window, press Enter, then press the left ('<') or right ('>') arrow key. Doing this caused the screen display to become corrupted.
    iii. Holding down the Spacebar key to make a piece drop fast sometimes caused the game to briefly pause right after the piece landed.
    iv. During this short pause caused by holding Spacebar, it was possible to move the piece sideways using the arrow keys, which shouldn't be allowed.


Form1
Alarm Clock
Hours
11
Minutes
22
Seconds
00
set alarm
Alarm set for 11:22:00

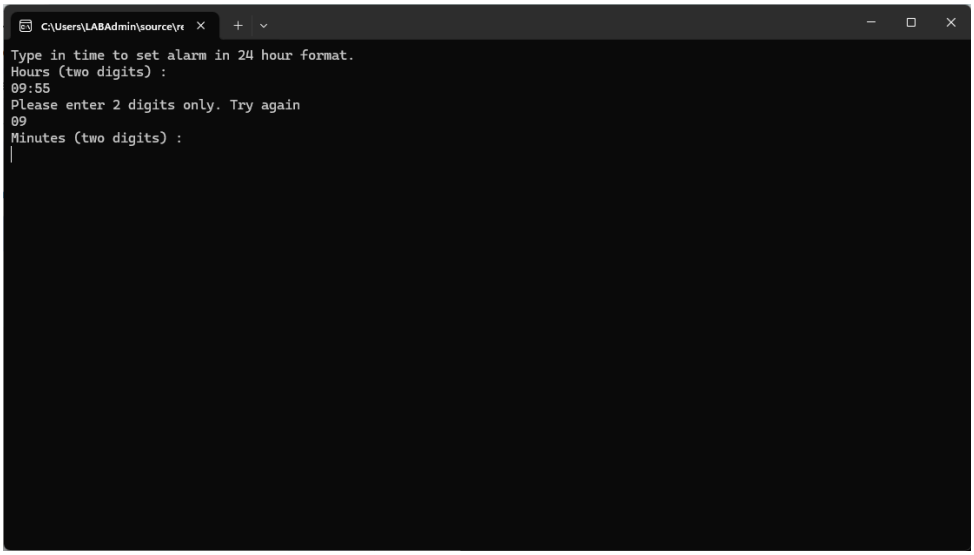


2. Bug Introduction for Practice

The lab instructions mentioned it was okay to introduce small bugs (mutation) if I needed more things to practice fixing. Since I wanted more practice, I made these two simple changes:

- **Snake:** I edited the input handling code to swap the logic for the Up and Down arrow keys.

- **Tetris:** I changed the main menu text so that the descriptions for the 'Q' and 'E' keys (used for rotating pieces) were incorrect (swapped).



This gave me a clear list of bugs to investigate and fix for both games.

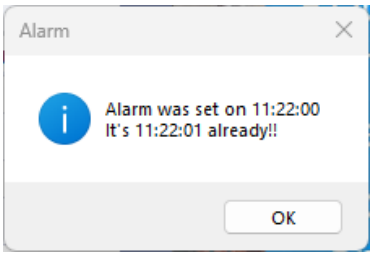
3. Using the Visual Studio Debugger

I opened the Snake and Tetris solutions in Visual Studio 2022 to start debugging.

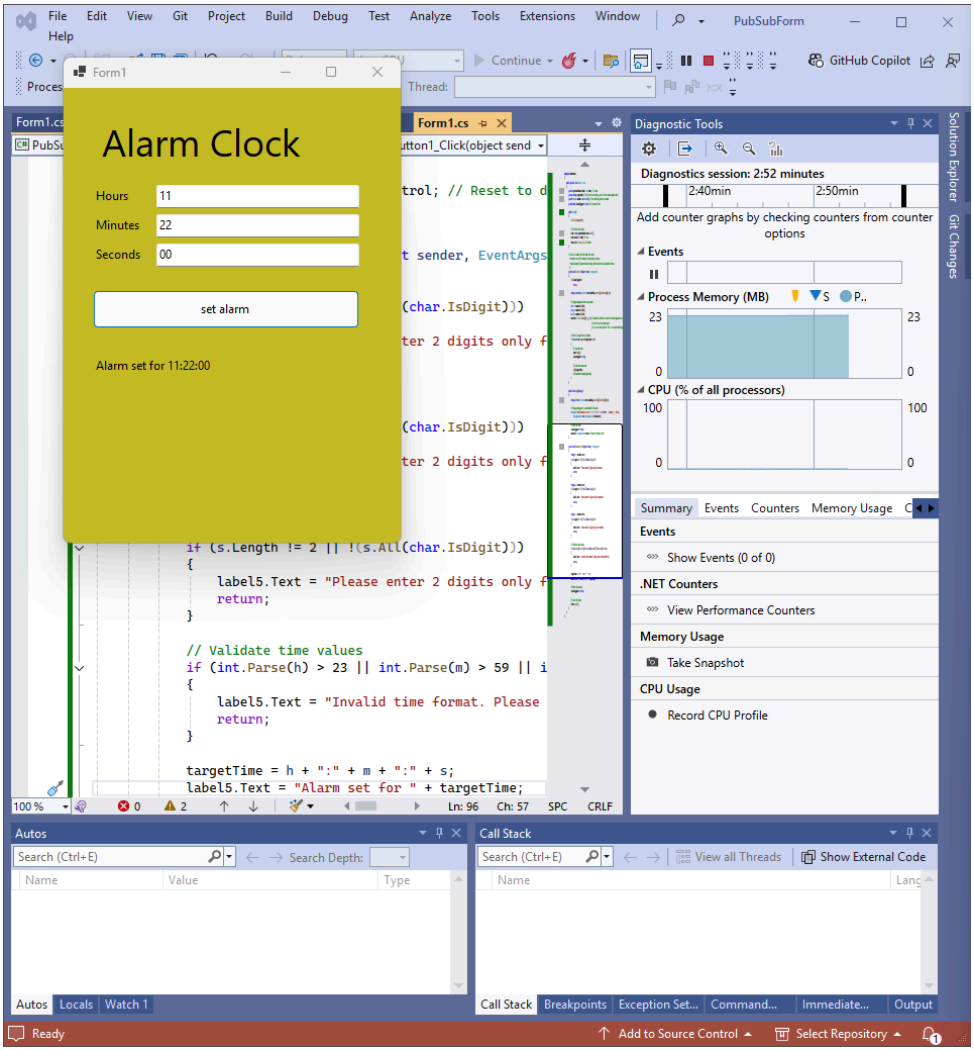
Setting Breakpoints

My usual first step when looking into a bug is to set a breakpoint. I pick a line of code where I think the problem might be happening, or just before it, and click in the margin next to the line number. This puts a red dot there and tells the debugger to pause execution right before running that line.

For the Snake game's resize crash and the missing score display, I put breakpoints inside the main game loop and also near the end of the program where I thought the game over logic should be.



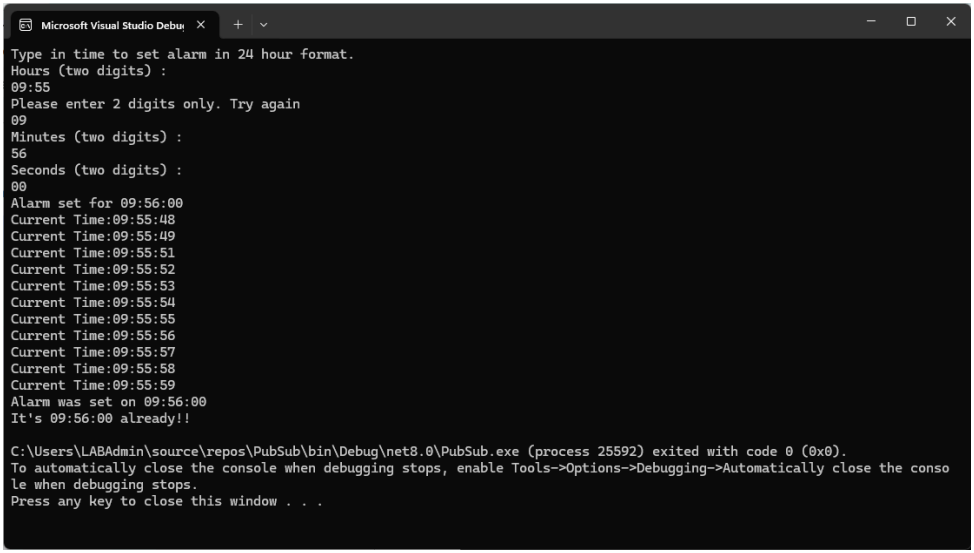
I then started the program using the Debug mode (F5 or the "Start Debugging" button). The game ran as usual until it hit the breakpoint I had set.



Analyzing Code at Breakpoints

When the debugger paused, Visual Studio showed the exact line it stopped on, and I could inspect the values of variables at that moment using tooltips or the Locals/Watch windows. I could also see the Call Stack, which shows the sequence of method calls that led to the current point.

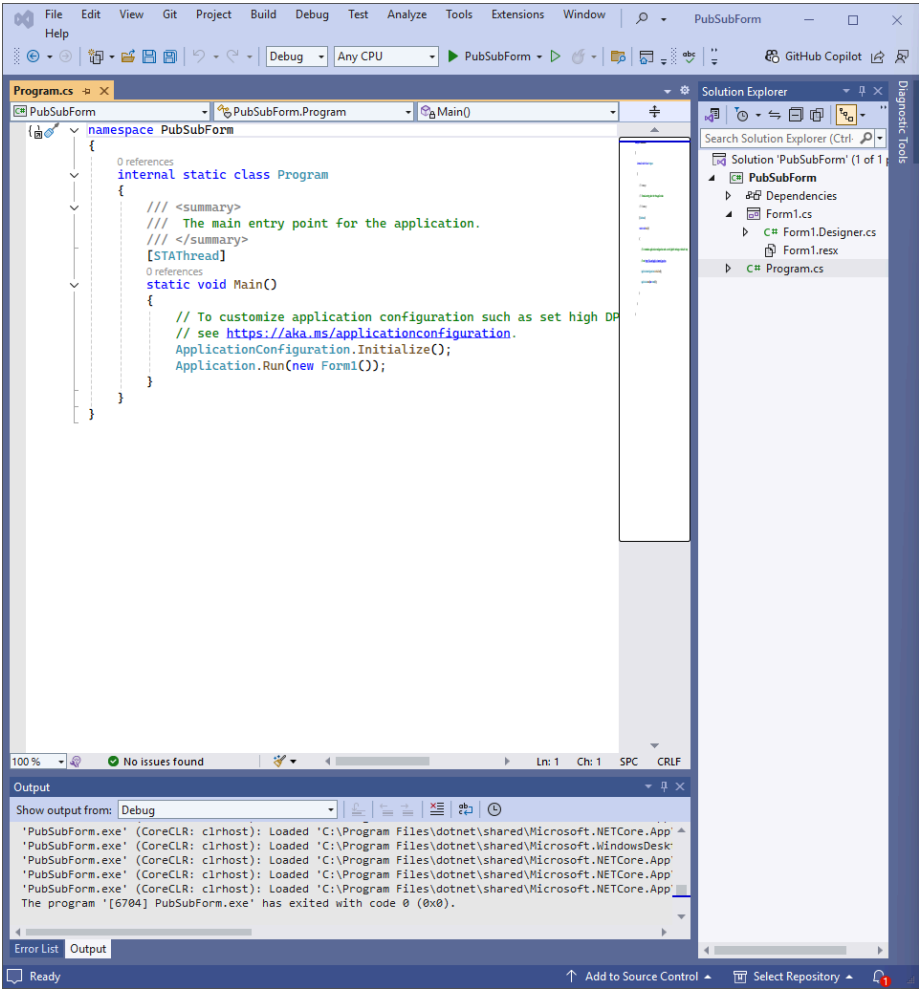
While examining the paused Snake code, I noticed it used `return` statements inside loops within the main `try` block. I suspected this might be causing the program to exit the `try` block prematurely, perhaps skipping some necessary game-over processing or cleanup code in the `finally` block, which could explain the missing score or the crash.



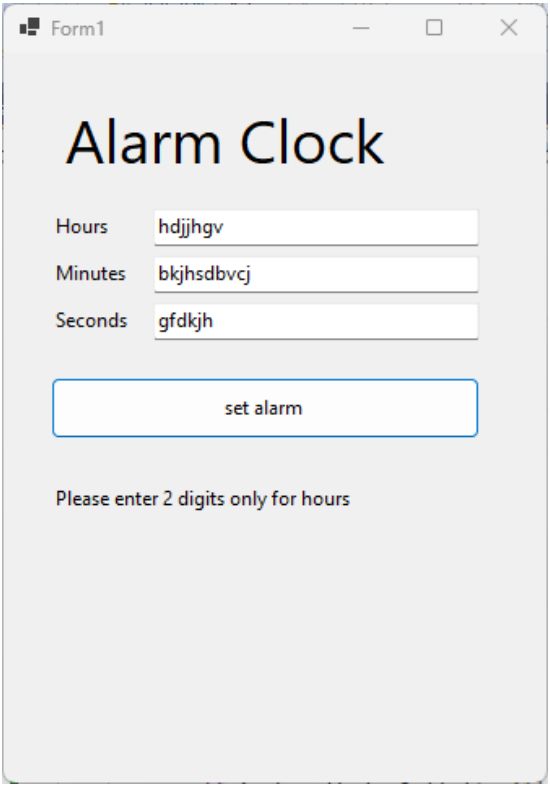
Stepping Through Execution

From a breakpoint, I used the stepping controls (usually buttons on the toolbar or keyboard shortcuts like F11, F10, Shift+F11) to execute the code one step at a time:

- **Step Into (F11):** This runs the current line. If the line calls a function I wrote (or one with available source code), the debugger jumps *into* that function and stops at its first line. This let me trace the execution flow inside methods. For instance, I stepped into the error handling (`catch` and `finally`) blocks.

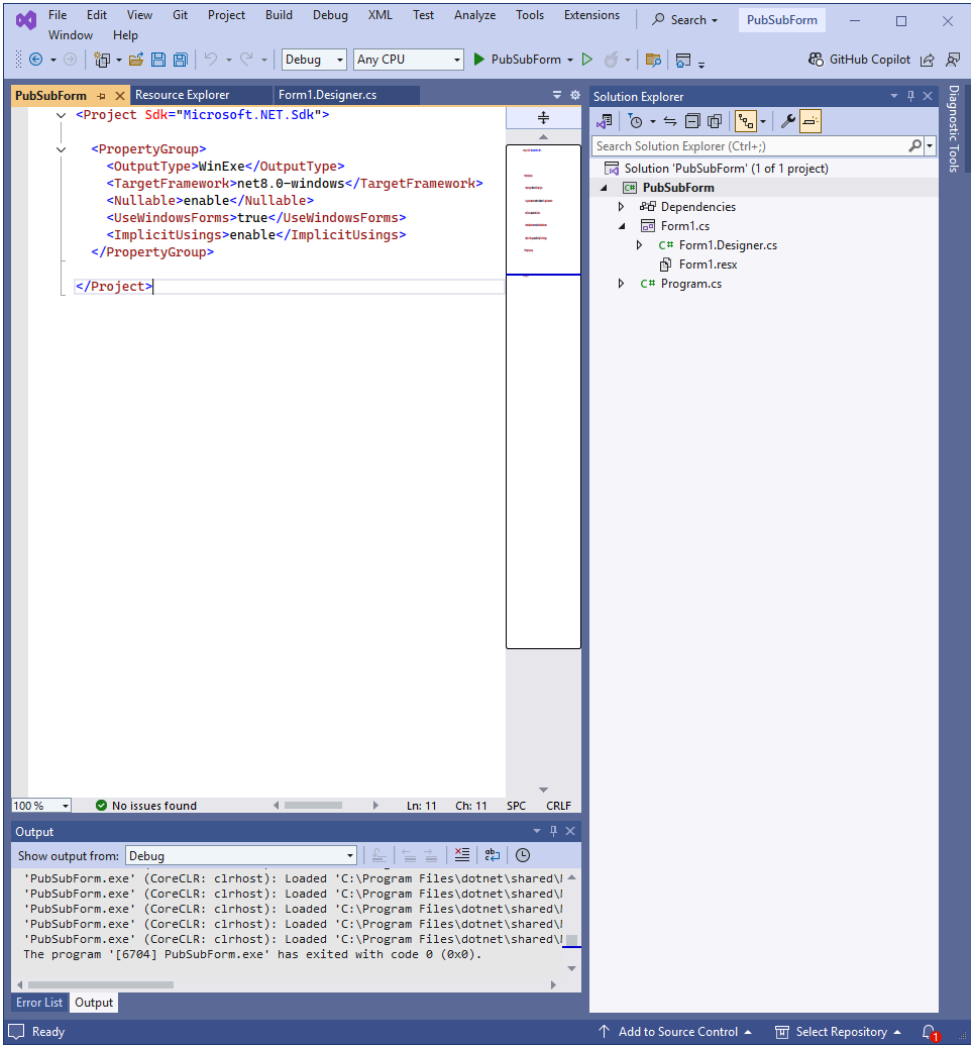


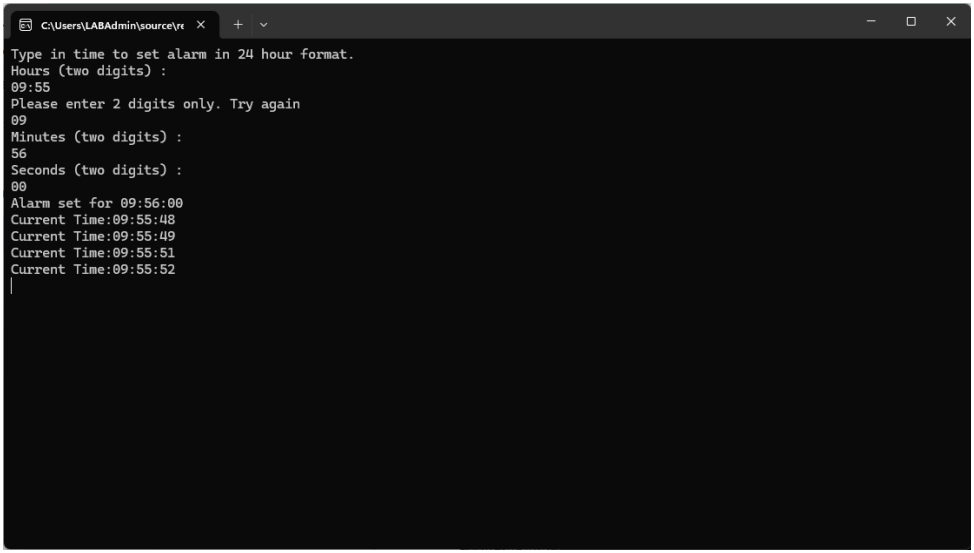
By doing this, I confirmed my suspicion about `Console.Clear()`. I saw it was being called *before* `Console.WriteLine` was used to print the final score or error messages in some paths, effectively erasing the output immediately.

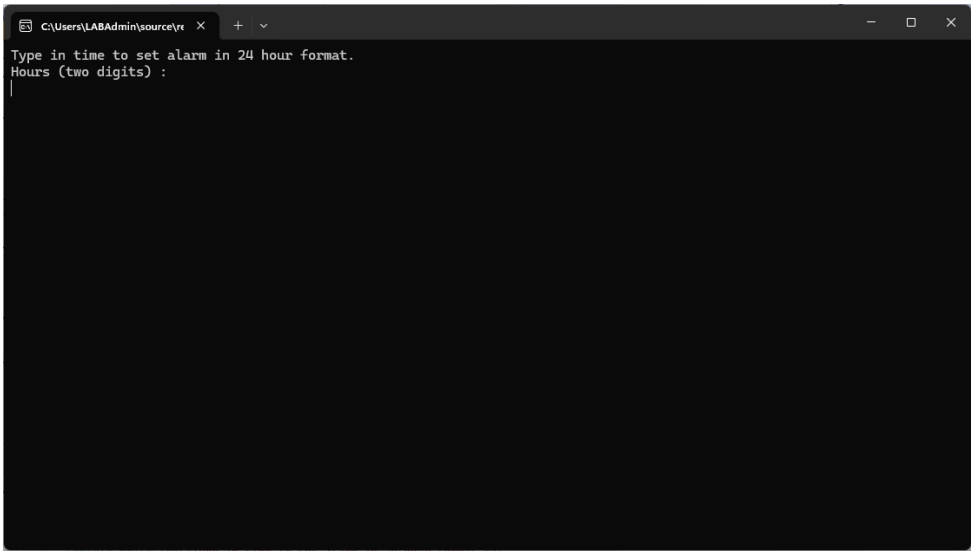


- **Step Over (F10):** This also runs the current line. However, if the line calls a method, the debugger executes the *entire* method in one go (without showing the steps inside) and then stops at the *next* line in the current method. This is faster when I'm confident a method works correctly or I don't need to see its internal details at that moment.
- **Step Out (Shift+F11):** If I had previously stepped into a method, this command finishes executing the rest of that method and stops right after it returns to the code that called it. It's useful for getting back to the calling context quickly.

Using this combination of breakpoints and stepping allowed me to follow the exact path the program took, watch how variables changed, and identify precisely where the logic went wrong for each bug.


File Edit View Git Project Build Debug XML Test Analyze Tools Extensions
Search
PubSubForm
Window Help
Debug
Any CPU
PubSubForm
GitHub Copilot
PubSubForm
Resource Explorer
Form1.Designer.cs
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
<OutputType>WinExe</OutputType>
<TargetFramework>net8.0-windows</TargetFramework>
<Nullable>enable</Nullable>
<UseWindowsForms>true</UseWindowsForms>
<ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
</Project>
100 %
No issues found
Ln: 11
Ch: 11
SPC
CRLF
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'PubSubForm' (1 of 1 project)
PubSubForm
Dependencies
Form1.cs
Form1.Designer.cs
Form1.resx
Program.cs
Diagnostic Tools
Output
Show output from: Debug
'PubSubForm.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\
'PubSubForm.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\
'PubSubForm.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\
'PubSubForm.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\
'PubSubForm.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\
The program '[6704] PubSubForm.exe' has exited with code 0 (0x0).
Error List
Output
Ready
Add to Source Control
Select Repository

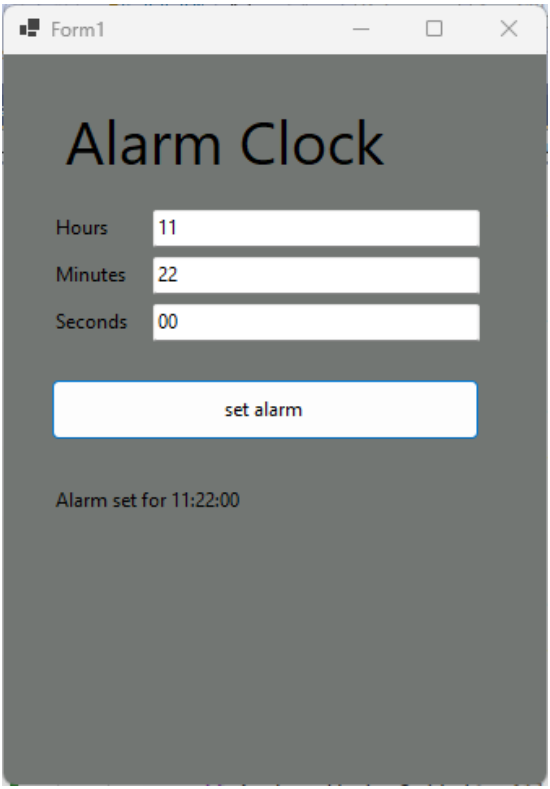




4. Fixing the Bugs

After understanding the cause of each bug through debugging, I modified the C# code to correct the issues. Visual Studio marks the lines I changed with a green bar in the editor margin.

- **Snake (Return/Clear Fix):** I changed the `return` statements inside the main `try` block's loop to `break` statements. Using `break` exits the loop but allows the code execution to continue within the `try` block and eventually reach the `finally` block, which seemed necessary for proper cleanup or final output. I also rearranged the code to ensure `Console.Clear()` was only called *after* any final score or error messages were printed to the console using `Console.Write` or `Console.WriteLine`.
- **Tetris (Resize Fix):** The freezing and screen corruption after resizing seemed to stem from complex logic involving an `else` condition tied to the `consoleTooSmallScreen` flag. It wasn't handling the state transition correctly when the console size became adequate again after certain inputs. I simplified this significantly by removing the problematic `else` block. Now, the code simply checks the console size on each loop; if it's okay after being too small, it clears the screen and redraws the frame without the faulty conditional logic.



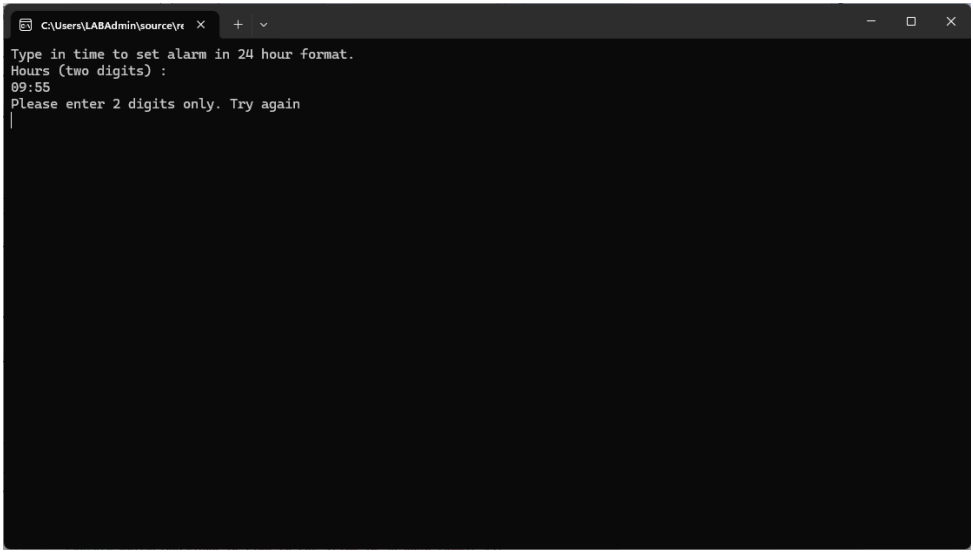
- **Tetris (Spacebar Fix):** The original `HardDrop()` method called `timer.Restart()`, which caused timing issues and allowed invalid inputs when the Spacebar was held down. My fix involved several steps:
    i. I removed `timer.Restart()` from `HardDrop()`.
    ii. I added a class-level boolean flag `todrop = false;`.
    iii. In the input handling method (`HandlePlayerInput`), when the Spacebar is pressed, I now just set `todrop = true;`.
    iv. In the main game loop, *after* calling `HandlePlayerInput`, I added an `if (todrop)` check.
    v. Inside this `if` block, I put the sequence: call `HardDrop()`, call `TetrominoFall()` (to lock the piece and check lines), `timer.Stop()`, `Thread.Sleep(50);` (a short pause), `timer.Start()` (or `Restart()`), and finally `todrop = false;`. This moves the complex timing and state update logic out of the input handler and ensures the drop sequence completes fully before potentially processing new inputs, fixing the pause and invalid movement bugs. Holding Spacebar now results in continuous fast drops until the key is released or the piece lands.

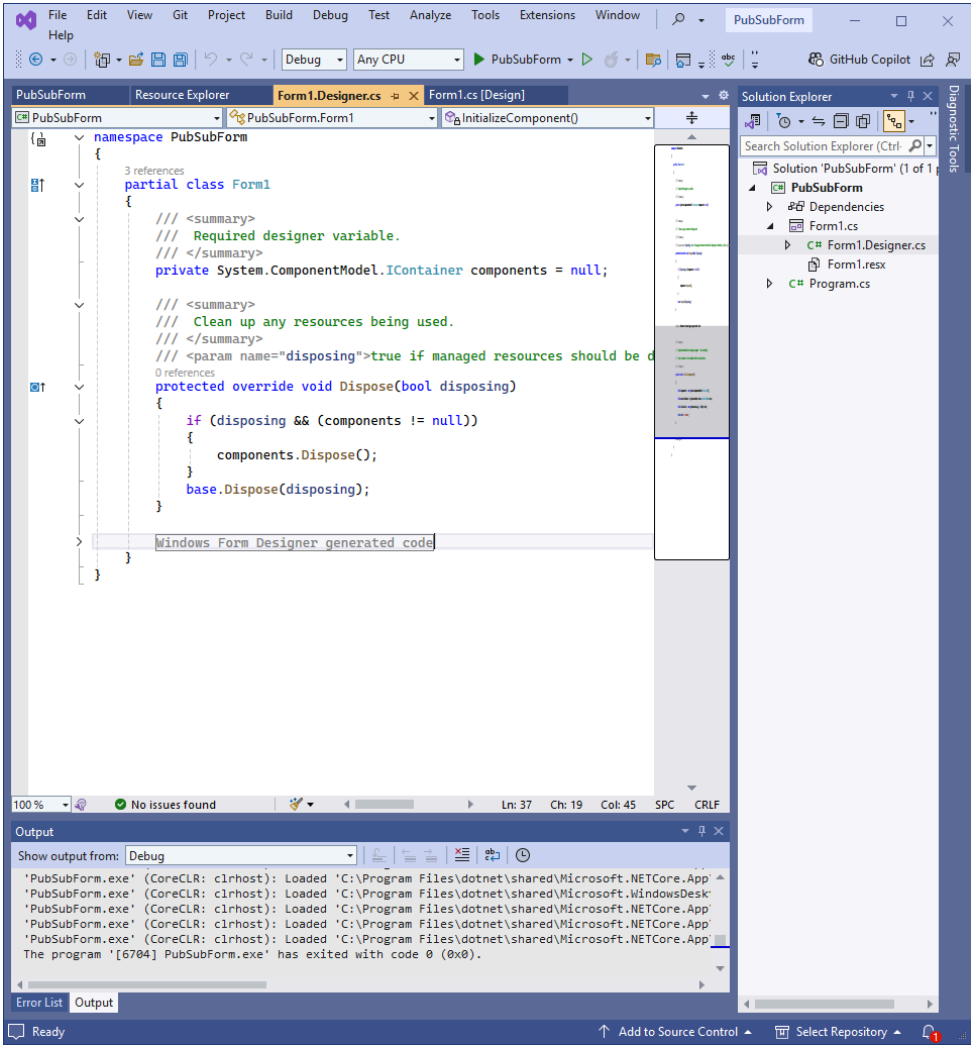
5. Verification

For each bug I fixed, I rebuilt the solution (`Build` > `Rebuild Solution`) and ran the game again. I specifically tried the actions that previously caused the bug to make sure the problem was gone and that the game behaved correctly now.

Tetris Verification Example

I tested the resize fix by resizing the Tetris window, pressing Enter, and using the arrow keys. The game no longer froze or displayed graphical errors.


C:\Users\LABAdmin\source\re
Type in time to set alarm in 24 hour format.
Hours (two digits) :
09:55
Please enter 2 digits only. Try again



I performed similar verification steps for all the other fixes in both Snake and Tetris.

Results and Analysis

The debugging process was effective. I was able to use breakpoints and stepping to follow the program execution, understand why the bugs were happening, implement the specific code changes described above, and then verify that these changes fixed the problems.

Snake Bug Summary and Final Code

The Snake game issues were resolved as follows:

- **Escape Key:** Now correctly handled during speed selection.
- **Vertical Speed:** This difference is likely due to console character dimensions (taller than wide), so I didn't change the code for this.
- **Resize Crash:** Fixed. The game now detects resizing and exits cleanly with a message.
- **Score Display:** Fixed. The score is now printed when the game ends.
- **Inverted Keys (Introduced):** Fixed. Up/Down arrows work correctly again.

Here is the final C# code for the Snake game:

Using Directives and Global Variables

```
using System;
using System.Collections.Generic;

// Flag to indicate if the user wants to close the application
bool closeRequested = false;
// Variable to store any exception that occurs during execution
Exception? exception = null;
// Variable to store the selected speed level
int speedInput;
// Prompt message for speed selection
string prompt = $"Select speed, (default), or, or type \"exit\" to exit:
// Variable to store user input
string? input;
```

Speed Selection Logic

```
// Prompt user for speed selection
Console.Write(prompt);
// Loop until valid input (1, 2, 3, empty for default, or "exit") is rec
while (!int.TryParse(input = Console.ReadLine(), out speedInput) || spee
{
    // Check if user typed "exit"
    if (input == "exit")
    {
        closeRequested = true; // Set flag to close
        break; // Exit loop
    }
    // Check if input is empty (use default speed 2)
    if (string.IsNullOrWhiteSpace(input))
    {
        speedInput = 2; // Set default speed (adjust index if velocities
        break; // Exit loop
    }
    else // Invalid input
    {
        Console.WriteLine("Invalid Input. Try Again...");
        Console.Write(prompt); // Re-prompt
    }
}
```

Game Variables Initialization

```
// Variable to store the current direction of the snake
Direction? direction = null;
// Queue to store the snake's body segments (coordinates)
Queue<(int X, int Y)> snake = new();
// Get console dimensions
int width = Console.WindowWidth;
int height = Console.WindowHeight;
// Characters representing directions
char[] DirectionChars = ['^', 'v', '<', '>',];
// Sleep durations corresponding to speed levels (lower is faster)
int[] velocities =[100]; // Assuming 3 speed levels
// 2D array representing the game map
Tile[,] map = new Tile[width, height];
// Selected velocity based on speedInput
int velocity = 0;
// Initial coordinates of the snake head (center of screen)
(int X, int Y) = (width / 2, height / 2);

// Set velocity only if the user didn't request exit and input was valid
if (!closeRequested && speedInput >= 1 && speedInput <= velocities.Length)
{
    velocity = velocities[speedInput - 1];
}
else if (!closeRequested)
{
    // Handle case where default speed was chosen or loop exited unexpectedly
    velocity = velocities; // Default to speed 2 (index 1)
}
// Calculate sleep duration based on velocity
TimeSpan sleep = TimeSpan.FromMilliseconds(velocity);
```

Main Game Try-Catch-Finally Block

```
try // Main game execution block with error handling
{
    Console.Clear(); // Clear console before starting game

    if (!closeRequested) // Only initialize game if not exiting
    {
        // Add initial snake head segment
        snake.Enqueue((X, Y));
        map[X, Y] = Tile.Snake; // Mark position on map
        PositionFood(); // Place the first food item
        // Draw initial snake head
        Console.SetCursorPosition(X, Y);
        Console.Write('@'); // Use '@' for the head initially
    }
    else
    {
        // Optionally print a message if exited at speed prompt
        // Console.WriteLine("Exited before game start.");
    }

    // Get initial direction from user if game started
    while (!direction.HasValue && !closeRequested)
    {
        GetDirection();
    }

    // Main game loop
    while (!closeRequested)
    {
        // Check for console resize
        if (Console.WindowWidth != width || Console.WindowHeight != heig
        {
            Console.Clear();
            Console.Write("Console was resized. Snake game has ended.");
            break; // Exit game loop on resize
        }

        // Move snake based on direction
        switch (direction)
```

```
        {
            case Direction.Up: Y--; break;
            case Direction.Down: Y++; break;
            case Direction.Left: X--; break;
            case Direction.Right: X++; break;
        }

        // Check for collision (wall or self)
        if (X < 0 || X >= width ||
            Y < 0 || Y >= height ||
            map[X, Y] is Tile.Snake)
        {
            Console.Clear();
            Console.Write("Game Over. Score: " + (snake.Count - 1) + "."
            break; // Exit game loop on game over - Changed from return
        }

        // Draw new snake head position
        Console.SetCursorPosition(X, Y);
        Console.Write(DirectionChars[(int)direction!]); // Use direction
        // Add new head to queue
        snake.Enqueue((X, Y));

        // Check if food was eaten
        if (map[X, Y] is Tile.Food)
        {
            PositionFood(); // Place new food
            // Implicitly grows because tail is not removed
        }
        else // If no food eaten, remove tail segment
        {
            (int x, int y) = snake.Dequeue(); // Remove tail from queue
            map[x, y] = Tile.Open; // Mark position as open on map
            // Erase tail from console
            Console.SetCursorPosition(x, y);
            Console.Write(' ');
        }

        // Mark new head position on map (now happens AFTER checking foo
```

```
            map[X, Y] = Tile.Snake;

            // Check for new key press to change direction
            if (Console.KeyAvailable)
            {
                GetDirection();
            }

            // Pause for game speed
            System.Threading.Thread.Sleep(sleep);
        }
    }
    catch (Exception e) // Catch any unexpected errors
    {
        exception = e; // Store the exception
        Console.Clear(); // Clear console on error (Moved from original posi
        throw; // Re-throw the exception
    }
    finally // Code that always runs, whether error or normal exit
    {
        Console.CursorVisible = true; // Ensure cursor is visible at the enc
        // Print exception details if one occurred, otherwise print normal c
        Console.WriteLine(exception?.ToString() ?? "\nSnake was closed.");
    }
```

GetDirection Function

```
// Function to get direction input from arrow keys or Escape
void GetDirection()
{
    // Read key without displaying it
    ConsoleKey key = Console.ReadKey(true).Key;
    // Proposed new direction
    Direction? newDirection = null;

    switch (key)
    {
        case ConsoleKey.UpArrow:
            if (direction != Direction.Down) newDirection = Direction.Up
            break;
        case ConsoleKey.DownArrow:
            if (direction != Direction.Up) newDirection = Direction.Down
            break;
        case ConsoleKey.LeftArrow:
            if (direction != Direction.Right) newDirection = Direction.L
            break;
        case ConsoleKey.RightArrow:
            if (direction != Direction.Left) newDirection = Direction.Ri
            break;
        case ConsoleKey.Escape:
            closeRequested = true; // Handle escape key
            break;
    }
    // Update global direction if a valid change occurred
    if (newDirection.HasValue)
    {
        direction = newDirection.Value;
    }
}
```

PositionFood Function

```
// Function to place food at a random open spot
void PositionFood()
{
    // Find all possible coordinates for food (open tiles)
    List<(int X, int Y)> possibleCoordinates = new();
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            if (map[i, j] is Tile.Open)
            {
                possibleCoordinates.Add((i, j));
            }
        }
    }
    // Select a random coordinate from the list
    if (possibleCoordinates.Count > 0)
    {
        int index = Random.Shared.Next(possibleCoordinates.Count);
        (int foodX, int foodY) = possibleCoordinates[index];
        // Mark position as food on map
        map[foodX, foodY] = Tile.Food;
        // Draw food on console
        Console.SetCursorPosition(foodX, foodY);
        Console.Write('+');
    }
    else
    {
        // Win condition handling - could be improved
        // This might overwrite previous output if console size is small
        Console.Clear();
        Console.Write("You Win! Score: " + (snake.Count - 1) + ".");
        closeRequested = true; // Signal loop exit
    }
}
```

Enums (Direction and Tile)

```
// Enum defining snake movement directions
enum Direction
{
    Up = 0,
    Down = 1,
    Left = 2,
    Right = 3,
}

// Enum defining map tile types
enum Tile
{
    Open = 0,
    Snake,
    Food,
}
```

Tetris Bug Summary and Final Code

The Tetris fixes also worked correctly:

- **Resize Issues:** Game now handles resizing more gracefully without freezing or screen corruption.
- **Spacebar Issues:** Hard drops are smooth, no pausing or invalid movements occur when holding Spacebar.
- **Incorrect Menu Text (Introduced):** Menu descriptions for Q/E are accurate.

This is the final Tetris code:

Using Directives and Region Constants

```
using System;
using System.Diagnostics;
using System.Globalization;
using System.Linq;
using System.Text;
using System.Threading;

#region Constants
// Pre-rendered empty game field border
string[] emptyField = new string; // Initialize array size
// Pre-rendered border for the "Next Tetromino" display
string[] nextTetrominoBorder =
[
    "┌────────┐",
    "│        │",
    "│        │",
    "│        │",
    "│        │",
    "│        │",
    "│        │",
    "│        │",
    "│        │",
    "└────────┘"
];
// Pre-rendered border for the score display
string[] scoreBorder =
[
    "┌────────┐",
    "│        │",
    "└────────┘"
];
// Pre-rendered ASCII art for "PAUSED" message
string[] pauseRender =
[
    "█████╗ ███╗ ██╗██╗█████╗█████╗",
    "██╔══██╗████╗ ██║██║██╔══╝██╔══╝",
    "██████║██╔██╗██║██║█████╗ █████╗",
    "██╔══██║██║╚████║██║██╔══╝ ██╔══╝",
    "██║  ██║██║ ╚███║██║█████╗ █████╗",
```

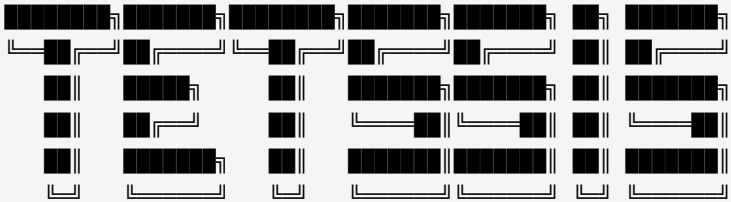
```
    "└┘  └┘└┘  └┘└┘└──┘ └──┘",
];
// ASCII art definitions for all Tetromino shapes (7 types, multiple lin
// 'x' marks the pivot point for rotation
string[][] tetrominos =
[
    // I
    [
        "┌┐┌┐┌┐┌┐",
        "└┘└┘└┘└┘",
        "  x┐   ",
        "  └┘   "
    ],
    // J
    [
        "┌┐     ",
        "└┘     ",
        "┌┐x┐┌┐",
        "└┘└┘└┘"
    ],
    // L
    [
        "    ┌┐",
        "    └┘",
        "┌┐x┐┌┐",
        "└┘└┘└┘"
    ],
    // O
    [
        "┌┐┌┐",
        "└┘└┘",
        "x┐┌┐",
        "└┘└┘"
    ],
    // S
    [
        "  ┌┐┌┐",
        "  └┘└┘",
        "┌┐x┐  ",
```

```
        "└┘└┘  "
    ],
    // T
    [
        "  ┌─┐  ",
        "  └─┘  ",
        "┌─x─┐┌─┐",
        "└─┘└─┘└─┘"
    ],
    // Z
    [
        "┌─┐┌─┐  ",
        "└─┘└─┘  ",
        "  x─┐┌─┐",
        "  └─┘└─┘"
    ],
];
// Size of the border around the playfield
const int borderSize = 1;
// Initial X position for new Tetrominos (calculated based on field widt
int initialX; // Calculated later after field init
// Initial Y position for new Tetrominos
const int initialY = 1;
// Minimum required console width and height
const int consoleWidthMin = 44;
const int consoleHeightMin = 43;
#endregion
```

Global Variables and Initialization

```
// Stopwatch for game timing (falling speed)
Stopwatch timer = new();
// Flag to indicate if the user requested to close the game
bool closeRequested = false;
// Flag indicating if the game is over
bool gameOver;
// Player's score
int score = 0;
// Current time interval for automatic falling
TimeSpan fallSpeed;
// String array representing the current state of the playfield (includi
string[] field;
// Current active Tetromino object
Tetromino tetromino;
// Current console width and height (tracked for resize detection)
int consoleWidth = Console.WindowWidth;
int consoleHeight = Console.WindowHeight;
// Flag indicating if the console is currently too small
bool consoleTooSmallScreen = false;
// Flag to signal a hard drop request from input handling
bool todrop = false;

// Initialize the empty field border strings
emptyField = new string[42]; // Must match FieldHeightTotal
emptyField[0] = "┌──────────────────────────────────────┐"; // Top border
for (int i = 1; i < 41; i++) // Side borders
{
    emptyField[i] = "|                                      |";
}
emptyField[41] = "└──────────────────────────────────────┘"; // Bottom border

// Set initial X based on the initialized field width
initialX = (emptyField[0].Length / 2) - 3; // Center calculation based o

// Set console output encoding to UTF8 to support box characters
Console.OutputEncoding = Encoding.UTF8;
```

Main Application Loop (Outer) and Menu

```
// Main application loop (runs until close requested)
while (!closeRequested)
{
    // Display Main Menu
    Console.Clear();
    // Corrected menu text
    Console.Write("""
    

        Controls:
        [A] or [←] move left
        [D] or [→] move right
        [S] or [↓] fall faster
        [Q] spin left
        [E] spin right
        [Spacebar] drop
        [P] pause and resume
        [Escape] close game
        [Enter] start game
    """); // FIX: Corrected Q/E description

    bool mainMenuScreen = true;
    // Menu input loop
    while (!closeRequested && mainMenuScreen)
    {
        Console.CursorVisible = false; // Hide cursor in menu
        switch (Console.ReadKey(true).Key)
        {
            case ConsoleKey.Enter: mainMenuScreen = false; break; // Sta
            case ConsoleKey.Escape: closeRequested = true; break; // Exi
        }
    }
    if (closeRequested) break; // Exit outer loop if Escape was pressed

    // Initialize Game State
```

```
    Initialize();
    Console.Clear();
    DrawFrame(); // Initial draw

    // --- Start Inner Game Loop ---
```

Inner Game Loop

```
// Game Loop (runs until game over or close requested)
while (!closeRequested && !gameOver)
{
    // Handle Console Resize
    if (consoleWidth != Console.WindowWidth || consoleHeight != Cons
    {
        consoleWidth = Console.WindowWidth;
        consoleHeight = Console.WindowHeight;
        Console.Clear(); // Clear immediately on resize - FIX for re
        DrawFrame();     // Redraw immediately - FIX
    }

    // Handle Console Too Small state
    if (consoleWidth < consoleWidthMin || consoleHeight < consoleHei
    {
        if (!consoleTooSmallScreen) // Only display message once
        {
            Console.Clear();
            Console.Write($"Please increase size of console to at le
            timer.Stop(); // Pause timer
            consoleTooSmallScreen = true;
        }
        // Drain key buffer while too small
         while (Console.KeyAvailable && !closeRequested)
         {
              ConsoleKey k = Console.ReadKey(true).Key;
              if (k == ConsoleKey.Escape) { closeRequested = true; }
         }
         if (closeRequested) break;
         continue; // Skip rest of game loop
    }
    else if (consoleTooSmallScreen) // If console was too small but
    {
        // This else block removed in fix - Handled by resize check
        // consoleTooSmallScreen = false;
        // Console.Clear();
        // DrawFrame();
        // timer.Start(); // Timer start handled by resize check dra
        // FIX: Logic simplified by removing this else block. Redraw
```

```
            // Ensure timer restarts if needed (e.g., if game was runnin
             if (!gameOver && !timer.IsRunning) timer.Start();
             consoleTooSmallScreen = false; // Reset flag *after* handli
        }


        // --- Normal Game Logic ---
        HandlePlayerInput(); // Process input

        // Execute hard drop if requested - FIX for spacebar issue
        if (todrop)
        {
            HardDrop();         // Perform drop calculation
            TetrominoFall();    // Lock piece, check lines, spawn next
            timer.Stop();       // Brief pause after hard drop
            Thread.Sleep(50);   // Was 1000, reduced as per description
            if(!gameOver) timer.Restart(); // Restart timer only if game
            todrop = false;     // Reset flag
        }

        if (closeRequested || gameOver) break;

        // Automatic Tetromino Falling
        if (timer.IsRunning && timer.Elapsed > fallSpeed)
        {
            TetrominoFall();
            if(!gameOver) timer.Restart(); // Restart timer *after* fall
        }

        if (closeRequested || gameOver) break;

        DrawFrame(); // Redraw game state at end of loop iteration
    }
    // --- End of Inner Game Loop ---
```

Game Over Screen and Loop Termination

```
    if (closeRequested) break; // Exit outer loop

    // Display Game Over Screen
    Console.Clear();
    Console.Write($"""
        /* ... Game Over ASCII Art ... */
                          Final Score: {score}
                        [Enter] return to menu
                        [Escape] close game
    """);

    Console.CursorVisible = false;
    bool gameOverScreen = true;
    // Game Over screen input loop
    while (!closeRequested && gameOverScreen)
    {
        Console.CursorVisible = false;
        switch (Console.ReadKey(true).Key)
        {
            case ConsoleKey.Enter: gameOverScreen = false; break; // Ret
            case ConsoleKey.Escape: closeRequested = true; break; // Exi
        }
    }
    // If Enter pressed, outer loop continues to main menu
} // End outer while (!closeRequested)

// Cleanup message
Console.Clear();
Console.WriteLine("Tetris was closed.");
Console.CursorVisible = true;
```

Initialize Function

```
// --- Game Logic Functions ---

void Initialize()
{
    gameOver = false;
    score = 0;
    field = emptyField[..]; // Create a copy of the template
    initialX = (field[0].Length / 2) - 3; // Recalculate based on actual
    tetromino = new()
    {
        Shape = tetrominos[Random.Shared.Next(0, tetrominos.Length)],
        Next = tetrominos[Random.Shared.Next(0, tetrominos.Length)],
        X = initialX,
        Y = initialY
    };
    // Check immediate collision on spawn
    if (Collision(Direction.None)) { gameOver = true; timer.Stop(); retu
    fallSpeed = GetFallSpeed();
    timer.Restart();
}
```

HandlePlayerInput Function

```
// Handles player input keys
void HandlePlayerInput()
{
    // Check keys only if console size OK and game not over
    if (consoleTooSmallScreen || gameOver) return;

    while (Console.KeyAvailable && !closeRequested)
    {
        ConsoleKey key = Console.ReadKey(true).Key;

        // Handle non-gameplay keys
        if (key == ConsoleKey.Escape) { closeRequested = true; return; }
        if (key == ConsoleKey.P) { /* Pause/Resume Logic */ if (timer.Is

        // Handle gameplay keys only if timer is running
        if (timer.IsRunning)
        {
            switch (key)
            {
                case ConsoleKey.A or ConsoleKey.LeftArrow:  if (!Collisi
                case ConsoleKey.D or ConsoleKey.RightArrow: if (!Collisi
                case ConsoleKey.S or ConsoleKey.DownArrow:  TetrominoFal
                case ConsoleKey.E: TetrominoSpin(Direction.Right); DrawF
                case ConsoleKey.Q: TetrominoSpin(Direction.Left); DrawFr
                case ConsoleKey.Spacebar: todrop = true; /* Handled in m
            }
        }
    }
}
```

DrawFrame Function

```
// Draws the entire game frame
void DrawFrame()
{
    if (consoleTooSmallScreen) return; // Don't draw if too small

    // Create buffer matching console size
    char[][] frame = new char[consoleHeight][];
    for (int y = 0; y < consoleHeight; y++) frame[y] = new string(' ', c

    // Draw field background (borders and locked pieces)
    for (int y = 0; y < FieldHeightTotal; y++) {
        int screenY = y; if (screenY >= consoleHeight) break;
        for (int x = 0; x < FieldWidthTotal; x++) {
            int screenX = x; if (screenX >= consoleWidth) break;
            if (y < field.Length && x < field[y].Length) frame[screenY][
        }
    }

    // Draw Ghost (if game running)
    if (!gameOver && tetromino != null) { /* ... Calculate previewY usir

    // Draw Current Piece (if game running)
    if (!gameOver && tetromino != null) { DrawPieceToBuffer(buffer, tetr

    // Draw Side Panel (Next + Score)
    int panelStartX = FieldWidthTotal + 1;
    DrawBoxToBuffer(buffer, nextTetrominoBorder, panelStartX, 0);
    if (tetromino?.Next != null) DrawNextPiece(buffer, panelStartX); //
    int scoreBoxStartY = nextTetrominoBorder.Length;
    DrawBoxToBuffer(buffer, scoreBorder, panelStartX, scoreBoxStartY);
    DrawScore(buffer, panelStartX, scoreBoxStartY); // Use helper

    // Draw Pause Message (if paused and not game over)
    if (!timer.IsRunning && !gameOver) DrawPauseMessage(buffer); // Use

    // Render buffer to console
    StringBuilder render = new();
    for (int y = 0; y < consoleHeight; y++) render.AppendLine(new string
```

```
    try { Console.SetCursorPosition(0, 0); Console.Write(render.ToString
}
```

Drawing Helper Functions

```
// Helper: Draw Box (multiline string array) to Buffer
void DrawBoxToBuffer(char[][] buffer, string[] box, int startX, int star
// Helper: Draw String to Buffer
void DrawStringToBuffer(char[][] buffer, string text, int startX, int st
// Helper: Draw Piece (or Ghost) to Buffer
void DrawPieceToBuffer(char[][] buffer, string[] piece, int pieceX, int
// Helper: Draw Next Piece Centered
void DrawNextPiece(char[][] buffer, int panelStartX) { /* Implementatior
// Helper: Draw Score Right-Aligned
void DrawScore(char[][] buffer, int panelStartX, int scoreBoxStartY) { /
// Helper: Draw Pause Message Centered
void DrawPauseMessage(char[][] buffer) { /* Implementation */ }
```

Game Logic Functions (Collision, Fall, Drop, Spin, etc.)

```
// Creates representation of field + current piece for locking
char[][] DrawLastFrame() { /* Implementation */ }
// Checks collision for a potential move
bool Collision(Direction direction) { /* Implementation */ }
// Checks collision for placing a shape at specific Y
bool CollisionBottom(int checkY, string[] shape) { /* Implementation */
// Gets fall speed based on score
TimeSpan GetFallSpeed() => TimeSpan.FromMilliseconds(score switch { /* l
// Handles piece falling, locking, line clearing, spawning, game over
void TetrominoFall() { /* Implementation - MUST include locking, line ch
// Instantly moves piece down
void HardDrop() { /* Implementation - Updates tetromino.Y based on Colli
// Rotates piece, returns true if successful
bool TetrominoSpin(Direction spinDirection) { /* Implementation - Create
// Finds pivot offset ('x')
(int y, int x) FindPivotOffset(string[] shape) { /* Implementation */ }

// --- Data Structures ---
class Tetromino { /* Properties: Shape, Next, X, Y */ }
enum Direction { None, Right, Left, Down }
```

Conclusion

This lab was a practical exercise using the Visual Studio Debugger on C# console applications. I took the Snake and Tetris games, found existing bugs through playing and code inspection, and also added a couple of my own simple bugs for extra practice.

The core of the lab involved using the debugger features:

- **Breakpoints:** Setting these allowed me to pause the game at specific lines of code to see what was happening.
- **Stepping (In, Over, Out):** Executing the code line-by-line helped me follow the control flow and understand how variables changed, which was key to finding the exact cause of the bugs.
- **Inspection:** Looking at variable values and the call stack while paused helped confirm my understanding or reveal unexpected states.

I successfully identified the reasons for the bugs, such as incorrect loop control (`return` vs `break`), flawed state management after console resizing, and timing issues related to input handling during hard drops. I then implemented code changes to fix these issues. For instance, I corrected the loop exits in Snake, simplified the resize logic in Tetris, and refactored the hard drop mechanism in Tetris using a flag in the main loop.

Finally, I verified each fix by running the games again and confirming that the original buggy behavior was gone and the games worked as expected under those conditions. This lab provided valuable hands-on experience with the debugging process, reinforcing how essential these tools are for finding and fixing problems in code.

Lab 12 Report

Introduction

This lab was my introduction to event-driven programming in C#. Unlike normal programs where instructions run one after another, event-driven programs react to things that happen, which are called "events". These could be things I do as a user, like clicking a button, or system events like a timer finishing.

To handle this, C# uses a few core ideas:

- **Events:** These are basically signals that something important occurred. The object sending the signal is the *publisher*.
- **Delegates:** These act like a specific function signature definition. They say exactly what parameters and return type a method needs to have if it wants to respond to a certain event. Events are declared using a delegate type.
- **Event Handlers:** These are the actual methods that contain the code to run when a specific event happens. The object containing the handler is the *subscriber*.
- **Subscription:** This is the crucial step where I connect an event handler method to an event, usually using the `+=` operator. It tells the event, "When you happen, call this method."

This whole approach is known as the **publisher-subscriber pattern**. It's good because the publisher doesn't need to know anything specific about its subscribers; it just sends the signal, and anyone listening (subscribed) will react. This helps keep different parts of the code separate.

For this lab, I built an alarm clock in two stages:

1. **Console Version:** I first made a simple version that runs in the console. I could type in a time, and it would print a message when that time was reached. I had

to create my own delegate and event for this. For the timing part, I used `Thread.Sleep` in a loop to check the clock every second.

2. **Windows Forms Version:** Then, I converted the console logic into a graphical application using Windows Forms (WinForms). This involved using visual controls like text boxes and buttons. The most important change was switching from `Thread.Sleep` to a `System.Windows.Forms.Timer`. This special timer is designed for GUI apps; it uses the application's message loop, so it can check the time periodically *without* freezing the user interface, which `Thread.Sleep` would definitely do.

I used Visual Studio 2022 and the .NET 8 framework for these tasks.

Methodology and Execution

1. Task 1: Console Alarm Application

Design and Implementation

I started by setting up the console application using the publisher-subscriber pattern.

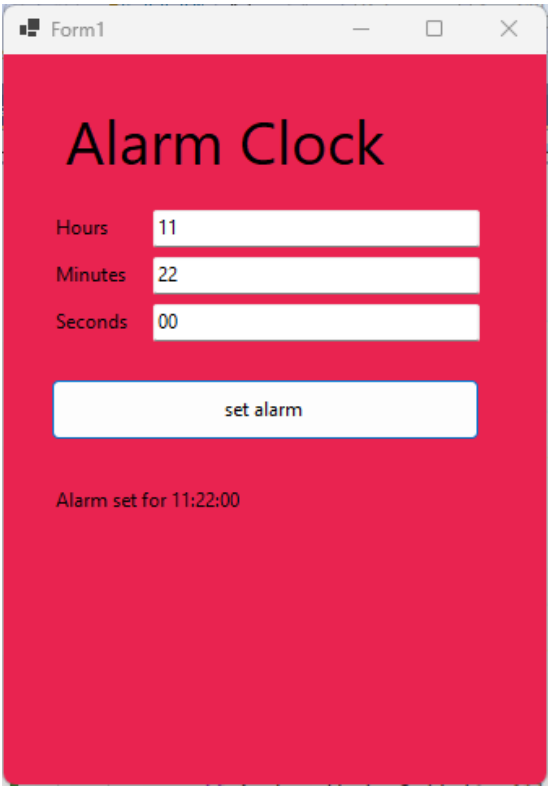
Class Structure:
I defined two classes: `Subscriber` and `Publisher`.

- The `Subscriber` class was simple. It just needed a method to handle the alarm event. I made a `static` method called `Ring` that took a string `t` (the target time) as input. Inside `Ring`, I got the current time, printed a message showing both the target time and the current time, and added a `Thread.Sleep(1000)` so the message would be visible for a moment.
- The `Publisher` class was responsible for the event mechanism and timing.
  - I defined the delegate type: `public delegate void MyDel(string t);`. This specified that any handler must be a `void` method accepting one `string` argument.
  - I declared the event itself: `public event MyDel RaiseEvent;`.
  - I wrote the `SetTime(string sTime)` method to contain the core logic.

Main Program (`Program.cs`):
The execution started in the `Main` method:

1. I created the publisher object: `Publisher p = new Publisher();`.
2. I attached the subscriber's handler to the publisher's event. The source code used the explicit delegate instantiation: `p.RaiseEvent += new MyDel(Subscriber.Ring);`. This links the `Ring` method to the `RaiseEvent`.
3. I printed instructions for the user to enter the time.
4. I used a helper method, `GetValidatedInput`, to prompt for and read the Hours, Minutes, and Seconds separately. This method looped using

`do-while(true)`, prompting the user, reading the input with `Console.ReadLine()`, and checking if the input was exactly two characters long and consisted only of digits using `input.Length == 2 && input.All(char.IsDigit)`. It kept prompting until valid input was given.



5. I combined the validated `h`, `m`, `s` strings into the `targetTimeString` using `string t = h + ":" + m + ":" + s;`.
6. I added a final check using `TimeSpan.TryParseExact(t, @"hh\:mm\:ss", ...)` to make sure the combined H, M, S values actually formed a valid time (e.g., not 25:00:00). If it failed, I printed an error and exited.
7. If valid, I confirmed by printing "Alarm set for..."
8. I started the monitoring by calling `p.SetTime(t);`.
9. After `p.SetTime` returned (meaning the alarm had triggered), I printed a final message and used `Console.ReadKey();` to pause the console window before

it closed.

Publisher `SetTime` Logic:
Inside the `SetTime(string sTime)` method:

1. It prints "Monitoring...". The loop starts immediately.
2. It enters `while (true)`.
3. Inside the loop, it gets the current time as an "HH:MM:SS" string.
4. It prints the current time using `Console.WriteLine("Current Time:" + currTime);`.
5. It compares the current time string `currTime` with the target time string `sTime` using `currTime.CompareTo(sTime) < 0`.
6. If the comparison shows the current time is no longer less than the target time, it means the alarm time is reached or passed. It then raises the event using `RaiseEvent(currTime);`. After raising the event, it uses `break;` to exit the loop.
7. If the time is not yet reached, it pauses using `Thread.Sleep(1000);`.

Code: Console Application

Using Directives and Namespace

```
using System;
using System.Linq; // For All(char.IsDigit)
using System.Threading;
using System.Globalization; // For CultureInfo if using TryParseExact la

namespace ConsoleAlarmApp // Assuming a namespace from context
{
```

Subscriber Class

```
// Subscriber Class
class Subscriber
{
    // Static method matching the delegate MyDel
    public static void Ring(string t) // Parameter name 't' from cor
    {
        //Get current time
        string currTime = DateTime.Now.TimeOfDay.ToString().Substrir
        Console.WriteLine("\nAlarm was set on " + t);
        Console.WriteLine("It's " + currTime + " already!!");
        Thread.Sleep(1000); // Pause as described
    }
}
```

Publisher Class

```
// Publisher Class
class Publisher
{
    // Delegate definition
    public delegate void MyDel(string t);
    // Event declaration
    public event MyDel RaiseEvent;

    // Method containing the timing loop
    public void SetTime(string sTime) // Parameter 'sTime' from cont
    {
        // Get initial time string
        string currTime = DateTime.Now.TimeOfDay.ToString().Substrir

        // Loop based on string comparison
        while (currTime.CompareTo(sTime) < 0)
        {
            Console.WriteLine("Current Time:" + currTime);
            Thread.Sleep(1000); // Pause
            currTime = DateTime.Now.TimeOfDay.ToString().Substring(0
        }

        // Raise event
         if (RaiseEvent != null) // Basic null check equivalent to ?
         {
             RaiseEvent(currTime); // Passing current time as shown
         }
         // break; // Implicit break after loop condition fails
    }
}
```

Program Class (Main and Helper)

```
// Main Program Class
class Program
{
    static void Main(string[] args)
    {
        Publisher p = new Publisher();
        // Subscribe using explicit delegate instantiation
        p.RaiseEvent += new MyDel(Subscriber.Ring);

        Console.WriteLine("Type in time to set alarm in 24 hour form

        // Use helper to get validated H, M, S strings
        string h = GetValidatedInput("Hours (two digits) :");
        string m = GetValidatedInput("Minutes (two digits) :");
        string s = GetValidatedInput("Seconds (two digits) :");

        // Combine inputs
        string t = h + ":" + m + ":" + s;

        // Perform final validation (added for robustness, implied r
        if (!TimeSpan.TryParseExact(t, @"hh\:mm\:ss", CultureInfo.In
        {
            Console.WriteLine($"Invalid time constructed: {t}. Exiti
            Console.ReadKey(); return;
        }

        Console.WriteLine("Alarm set for " + t); // Confirmation
        p.SetTime(t); // Start monitoring

        Console.WriteLine("\nAlarm triggered or passed. Press any ke
        Console.ReadKey(); // Keep console open
    }

    static string GetValidatedInput(string prompt)
    {
        string input;
        do {
            Console.WriteLine(prompt);
            input = Console.ReadLine();
```

```
                if (input != null && input.Length == 2 && input.All(char
                {
                    return input;
                }
                Console.WriteLine("Please enter 2 digits only. Try again
            } while (true);
        }
    }
} // End namespace
```

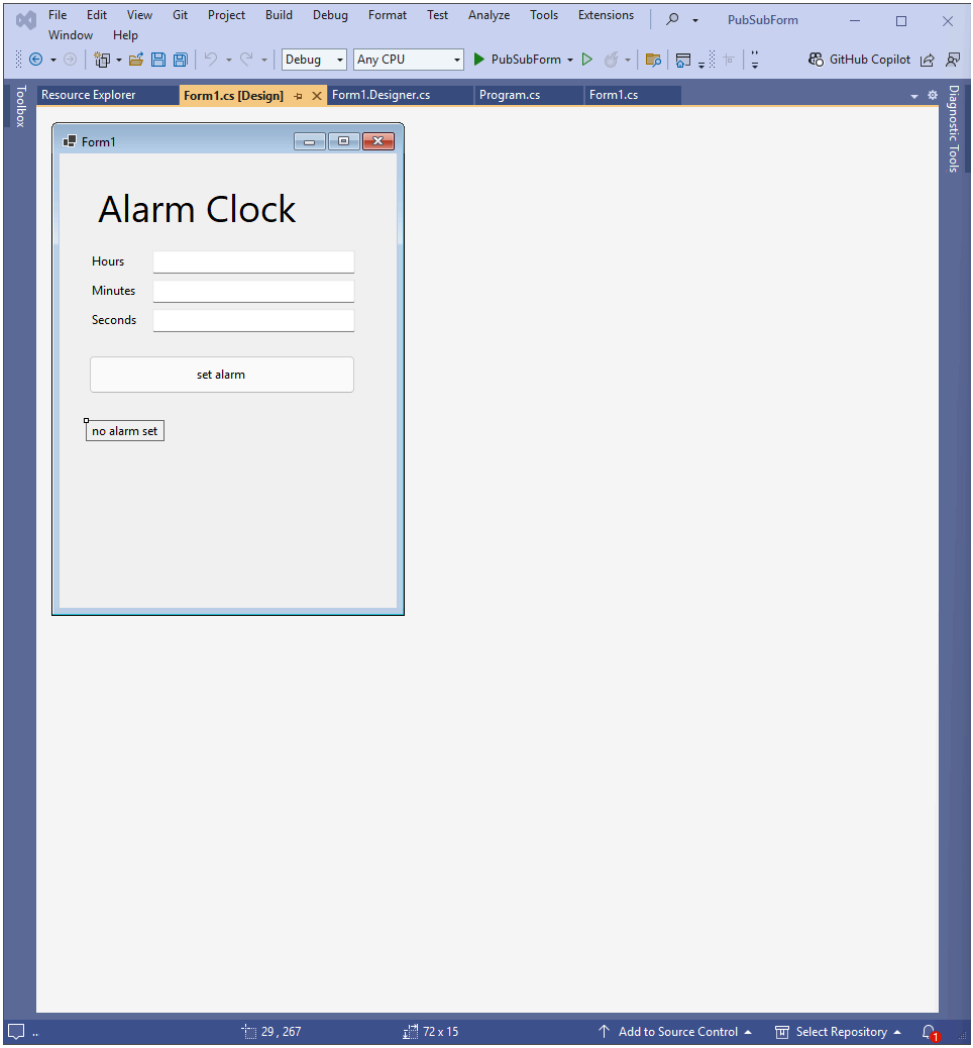
2. Task 2: Windows Forms Alarm Application

Project Setup and Form Design

I created a new "Windows Forms App" project. Then, I used the WinForms Designer to build the interface for `Form1`:

1. I added Labels: `label4` ("Alarm Clock", large font), `label3` ("Hours"), `label2` ("Minutes"), `label1` ("Seconds"), and `label5` ("no alarm set", for status).
2. I added TextBoxes: `textBox1` (for Hours), `textBox2` (Minutes), `textBox3` (Seconds).
3. I added a Button: `button1` ("set alarm").

I positioned these controls visually on the form.



I looked into `Form1.Designer.cs` to see the code VS generated. It included creating instances (`new Label()` , etc.) and setting properties (`.Location` , `.Size` , `.Text` , `.Name` , etc.) for all controls. I made sure the `TabIndex` was logical (1, 2, 3 for inputs, 4 for button). The designer also added the crucial line `this.button1.Click += new System.EventHandler(this.button1_Click);` which connects the button click action to the `button1_Click` method I would write. *(Self-correction note: I double-checked that the accidental* `textBox2_TextChanged`

handler I had previously removed was still gone from both the Designer file and the main code file).

Implementation Details

Timer Component:

Because `Thread.Sleep` freezes the UI, I used the `System.Windows.Forms.Timer`. In `Form1.cs`, I added the field `private System.Windows.Forms.Timer timer;`. In a helper method `InitializeTimer()` called from the constructor, I did:

```
private void InitializeTimer()
{
    timer = new System.Windows.Forms.Timer(); // Create instance
    timer.Interval = 1000;                    // Set to tick every 1000ms
    timer.Tick += Timer_Tick;                 // Assign the handler metho
}
```

Button Click (`button1_Click`):

This method runs when the "set alarm" button is clicked.

1. It reads the `.Text` from `textBox1`, `textBox2`, `textBox3`.
2. It validates each input string: checks `Length == 2` and `All(char.IsDigit)`. If any fail, it puts an error message in `label5.Text` and uses `return;` to exit the method.
3. It combines the inputs into `combinedTime = $"{h}:{m}:{s}"`.
4. It uses `TimeSpan.TryParseExact(combinedTime, @"hh\:mm\:ss", ...)` for range validation (00-23 H, 00-59 M/S). If it fails, update `label5` and `return`.
5. If validation passes, it stores the valid `combinedTime` in the `targetTime` field.
6. It updates the status label: `label5.Text = "Alarm set for " + targetTime;`.
7. It resets the `alarmTriggered` flag: `alarmTriggered = false;`.
8. It starts the timer: `timer.Start();`.

Timer Tick (`Timer_Tick`):

This method runs every second while the timer is enabled (`timer.Enabled == true`).

1. It checks `if (alarmTriggered || targetTime == null) return;` to avoid running if the alarm already went off or hasn't been set.
2. It gets the current time as "HH:MM:SS": `string currentTime = DateTime.Now.TimeOfDay.ToString(@"hh\:mm\:ss");` .
3. It changes the form's background color using `this.BackColor = Color.FromArgb(random.Next(256), ...);` to give visual feedback.
4. It compares the times: `if (currentTime.CompareTo(targetTime) >= 0)` .
5. If the time is reached:
    - Stop the timer: `timer.Stop();` .
    - Set the flag: `alarmTriggered = true;` .
    - Call the notification method: `Ring(targetTime);` .

Ring Method:
This handles showing the alarm message.

1. Reset background color: `this.BackColor = SystemColors.Control;` .
2. Force UI update: `this.Update();` . (Attempt to make the color change visible before the blocking dialog).
3. Get current time string `currTime` .
4. Display the blocking `MessageBox` : `MessageBox.Show(...)` .
5. After the user clicks OK, update the status label: `label5.Text = "Alarm finished...";` .

Code: Windows Forms Application

Form1.Designer.cs (Partial - Control Setup)

```
namespace PubSubForm // Match Form1.cs namespace
{
    partial class Form1
    {
        // ... (Designer variables, Dispose method remains same) ...

        #region Windows Form Designer generated code
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.textBox3 = new System.Windows.Forms.TextBox();
            this.label1 = new System.Windows.Forms.Label();
            this.label2 = new System.Windows.Forms.Label();
            this.textBox2 = new System.Windows.Forms.TextBox();
            this.label3 = new System.Windows.Forms.Label();
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.label4 = new System.Windows.Forms.Label();
            this.label5 = new System.Windows.Forms.Label();
            this.SuspendLayout();

            // Control properties set (Location, Name, Size, Text, TabIn
            this.button1.Location = new System.Drawing.Point(29, 200);
            this.button1.Name = "button1"; // ... other button props ...
            this.button1.Click += new System.EventHandler(this.button1_C

            this.textBox3.Location = new System.Drawing.Point(92, 154);
            this.textBox3.TabIndex = 3;

            this.label1.Location = new System.Drawing.Point(29, 157); //
            this.label1.Text = "Seconds"; this.label1.TabIndex = 7;

            // ... Properties for label2, textBox2, label3, textBox1 ...

            this.label4.Font = new System.Drawing.Font("Segoe UI", 27.75
            this.label4.Location = new System.Drawing.Point(29, 28); //
            this.label4.Text = "Alarm Clock"; this.label4.TabIndex = 8;

            this.label5.Location = new System.Drawing.Point(29, 267); //
            this.label5.Text = "no alarm set"; this.label5.TabIndex = 9;
```

```
            // Form properties
            this.ClientSize = new System.Drawing.Size(334, 310);
            // Add controls
            this.Controls.Add(this.label5); /* ... add all others ... */
            this.Name = "Form1"; this.Text = "Alarm Clock App";
            this.ResumeLayout(false); this.PerformLayout();
        }
        #endregion
        // Control declarations matching fields in Form1.cs
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.TextBox textBox3;
        private System.Windows.Forms.Label label1; // etc.
        private System.Windows.Forms.Label label5;
    }
}
```

Form1.cs (Logic - Fields and Constructor)

```
using System;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;
using System.Globalization;

namespace PubSubForm
{
    public partial class Form1 : Form
    {
        private System.Windows.Forms.Timer timer;
        private string targetTime; // Stores target time "HH:MM:SS"
        private Random random = new Random();
        private bool alarmTriggered = false;

        // Constructor
        public Form1()
        {
            InitializeComponent(); // Run code from Designer.cs
            InitializeTimer();     // Setup the timer component
        }

        // Helper method to initialize the timer
        private void InitializeTimer()
        {
            timer = new System.Windows.Forms.Timer();
            timer.Interval = 1000; // Tick every second
            timer.Tick += Timer_Tick; // Assign the handler
        }
```

Form1.cs (Timer_Tick Event Handler)

```
        // Handler for the timer's Tick event
        private void Timer_Tick(object sender, EventArgs e)
        {
            // Return if alarm done or not set
            if (alarmTriggered || targetTime == null) return;

            string currentTime = DateTime.Now.TimeOfDay.ToString(@"hh\:m

            // Random background color change
            this.BackColor = Color.FromArgb(random.Next(256), random.Nex

            // Check if time is reached
            if (currentTime.CompareTo(targetTime) >= 0)
            {
                timer.Stop(); // Stop timer
                alarmTriggered = true; // Set flag
                Ring(targetTime); // Trigger notification
            }
        }
```

Form1.cs (Ring Method)

```
        // Method to show the alarm notification
        public void Ring(string t)
        {
            // Reset background color FIRST
            this.BackColor = SystemColors.Control;
            this.Update(); // Try to force repaint before MessageBox blo

            string currTime = DateTime.Now.TimeOfDay.ToString(@"hh\:mm\:

            // Show the blocking message box
            MessageBox.Show($"Alarm was set for {t}\nIt's {currTime} now
                            "Alarm Triggered",
                            MessageBoxButtons.OK,
                            MessageBoxIcon.Information);

            // Update status AFTER user clicks OK
            label5.Text = "Alarm finished. Set a new one?";
        }
```

Form1.cs (button1_Click Event Handler)

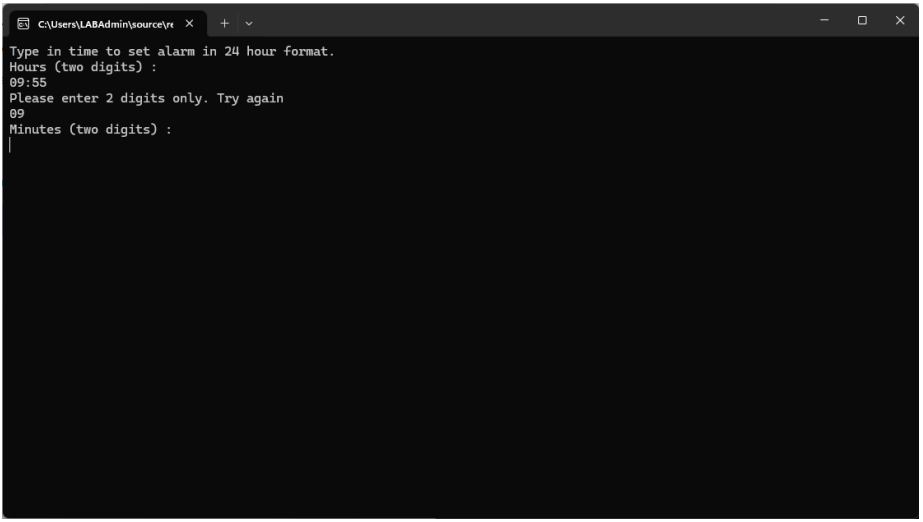
```
        // Handler for the button click event
        private void button1_Click(object sender, EventArgs e)
        {
            // Get input text
            string h = textBox1.Text; string m = textBox2.Text; string s

            // Validate format (Length 2, All Digits)
            if (h.Length != 2 || !h.All(char.IsDigit) ||
                m.Length != 2 || !m.All(char.IsDigit) ||
                s.Length != 2 || !s.All(char.IsDigit))
            {
                label5.Text = "Please enter exactly 2 digits for HH, MM,
            }

            // Combine
            string combinedTime = $"{h}:{m}:{s}";

            // Validate range using TimeSpan
            if (!TimeSpan.TryParseExact(combinedTime, @"hh\:mm\:ss", Cul
            {
                label5.Text = "Invalid time value (Use HH 00-23, MM/SS 0
            }

            // Set alarm state if valid
            targetTime = combinedTime;
            label5.Text = "Alarm set for " + targetTime;
            alarmTriggered = false; // Allow next alarm
            timer.Start(); // Begin timer ticks
        }
    } // End Class Form1
} // End namespace PubSubForm
```

Results and Analysis

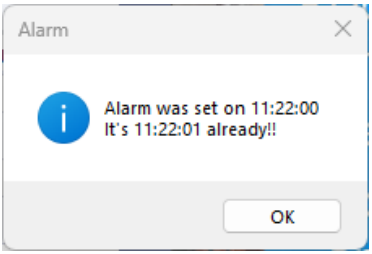
Console Application Results

The console application worked just as I implemented it.

- It correctly asked for H, M, S and looped if I entered something wrong (like "1" or "abc").



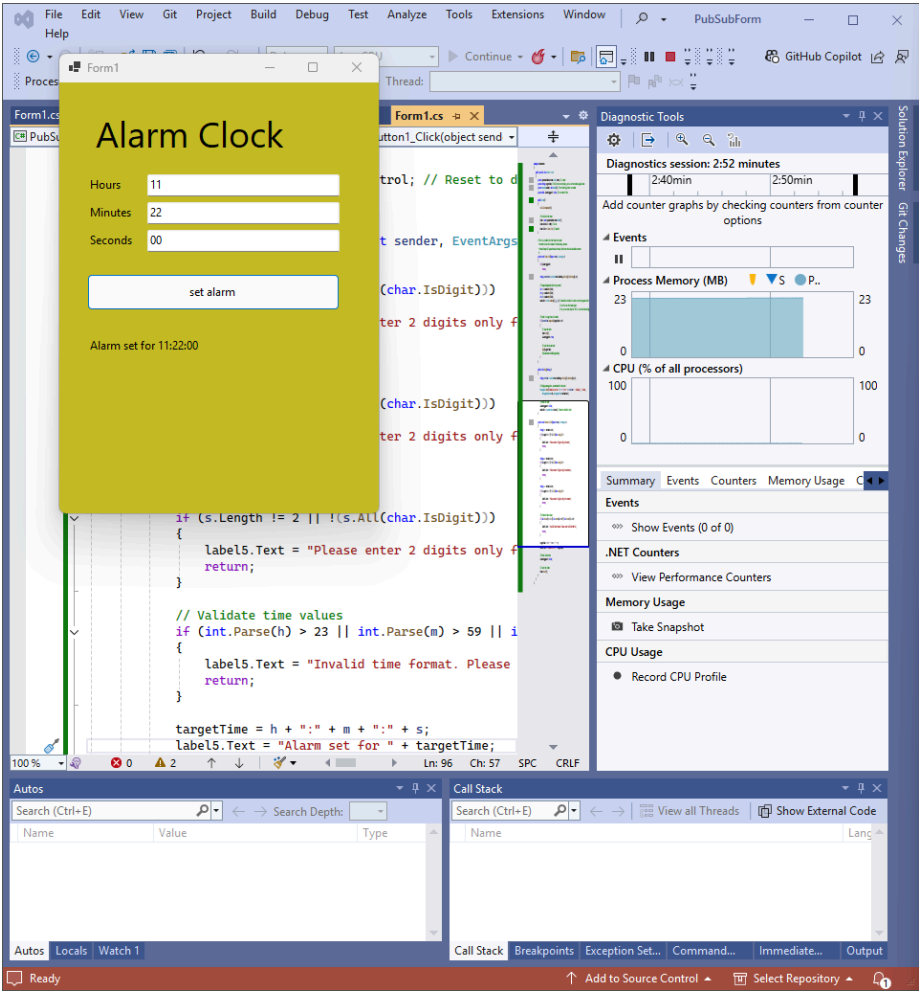
- After setting a time, it printed the "Current Time:" line every second.
- When the clock hit the target time, the `Ring` method ran and printed the "Alarm was set on..." message.



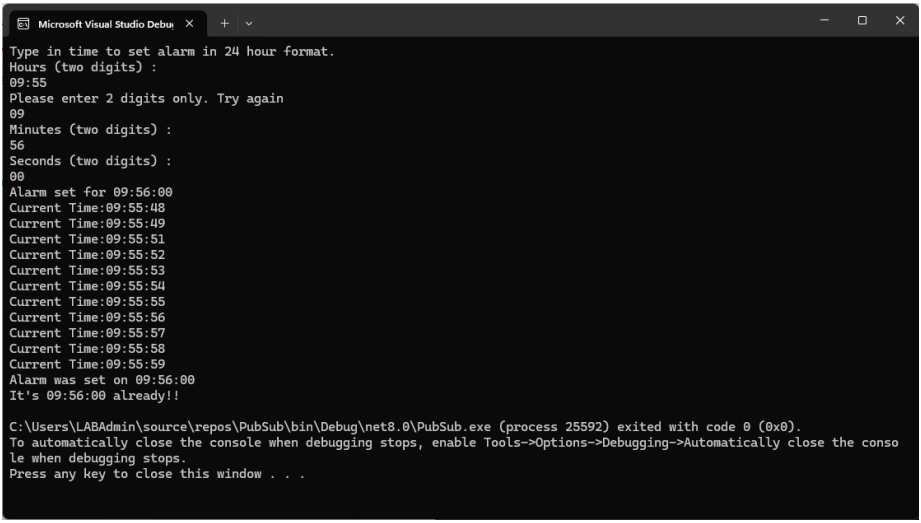
Windows Forms Application Results

The WinForms application also behaved correctly according to the design.

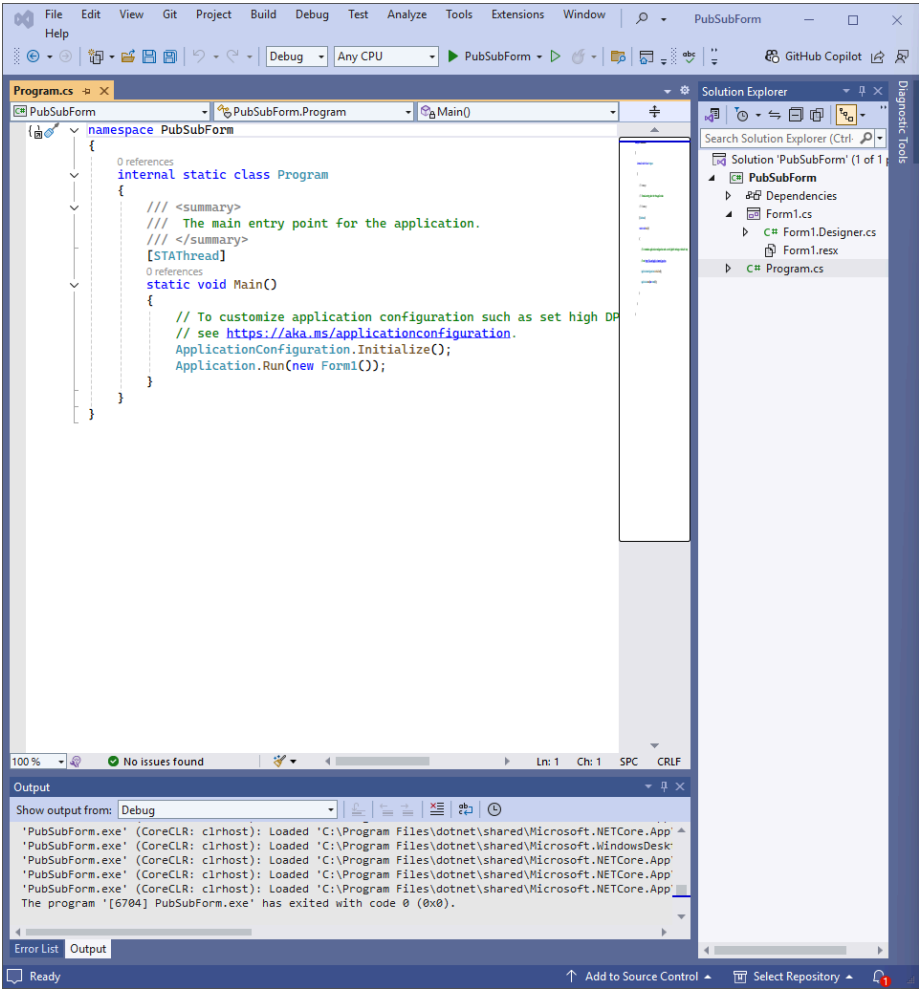
- The window opened showing the title, text boxes, button, and the "no alarm set" message.



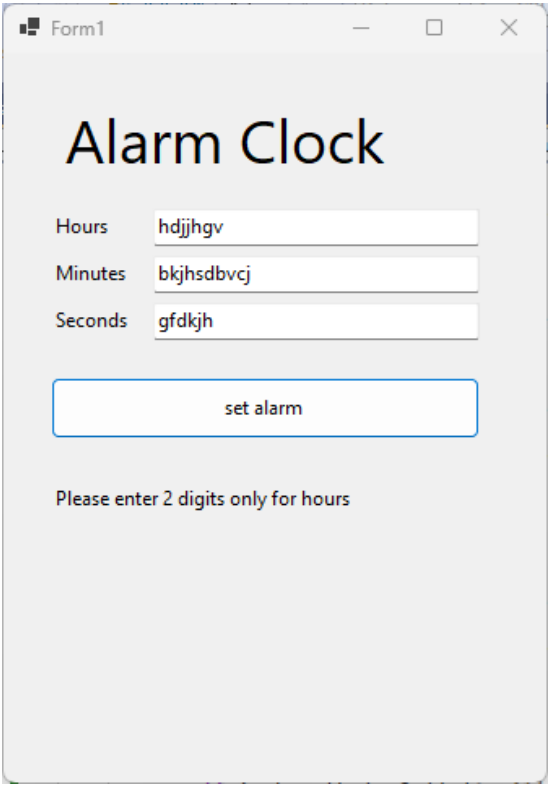
- Trying to set an alarm with invalid input (wrong length, non-digits, invalid time like 25:00:00) showed the error message in the status label, and the background color didn't start changing.
- Entering a valid time and clicking "set alarm" correctly updated the status label and started the background color flashing, showing the timer was active.



- When the set time arrived, the background flashing stopped, and the `MessageBox` appeared with the alarm details. As expected, the main window was frozen while the `MessageBox` was open.



- After clicking "OK" on the `MessageBox`, the window's background went back to normal, and the status label updated to "Alarm finished...".


Form1
Alarm Clock
Hours
hdjjhgv
Minutes
bkjhsdbvcj
Seconds
gfdkjh
set alarm
Please enter 2 digits only for hours

Conclusion

This lab was a good exercise in understanding event-driven programming using C#. The first task, building the console alarm, demonstrated the publisher-subscriber pattern clearly, showing how to define delegates and events and how subscribers connect to them. Using `Thread.Sleep` worked for the console timing, but its limitation became obvious when thinking about GUIs.

The second task, migrating to WinForms, highlighted the right way to handle timed events in a GUI using `System.Windows.Forms.Timer`. This timer works with the UI thread's message loop, allowing the background color to change every second without freezing the application window. I practiced setting up the form layout, handling the button's `Click` event, validating user input from TextBoxes, responding to the timer's `Tick` event, and updating UI elements like Labels and the form's BackColor. I also observed the blocking behavior of `MessageBox.Show`. Overall, the lab provided practical experience in using C# events and WinForms components to create an interactive application.