AC22005 – C# THREADS

OBJECTIVES

In this practical you will be required to build on your existing knowledge of C# and extend it to include threads. Threading is a central part of programming; it gives your program the ability to do more than one thing at once, which is crucial in developing responsive and efficient applications.
These exercises will be helpful in your development of the last coursework for this module.

GLOSSARY

Thread: a function that has the capacity to execute in parallel with other functions.

Multi-threaded: a program with multiple threads of execution.

Semaphore: a variable which is used for synchronisation between threads and for sharing resources. A special characteristic of the semaphore is that the operations that increase or decrease the value are "atomic" which means they cannot be interrupted.

Event handler: When the user does something in Windows (such as clicking a button), that becomes a Windows message. A Windows program can provide an event handler, which is a function that performs the action associated with that message.

PRACTICAL INSTRUCTIONS

The program you will write is going to simulate a horse race - we shall have **Red Rum** and **Desert Orchid** who will each execute in a thread of their own and race across your screen. There are several things we need to do to achieve this:

- create a Windows form application
- include some libraries for 2D graphics
- produce some GUI button controls to start the race and reset the race
- write the event handlers for the buttons
- create a couple of threads for the 'horses'
- write the threaded functions for the horses
- place bets and win!

ENTERING THE C# CODE

Start MS Visual Studio and create a new C# Windows Forms Application. Select the **program.cs** code window.
Cut and paste code sections from this document into the code window at the appropriate places. Firstly, you will need to replace the code stub with the following:

```
using System;
using System.Threading;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

/// <summary>
/// Uses threads to run an animation independently of controls
/// </summary>

public class Animation : Form
{

  // INSERT UI AND VARIABLES HERE

  public Animation()
    {
        // INSERT CONSTRUCTOR CODE HERE;
    }

// Main - this code always at the end
    public static void Main()
    {
        Application.Run(new Animation());
    }
}
```

As in previous examples, you will need to declare all your references to the UI components - use three buttons, one to start the race, one to pause and one to quit. We need some variables too:

```
private Panel controlPanel, drawingPanel;
private Button pauseButton, clearButton, quitButton;
private Thread DesertOrchid_t, RedRum_t;
private Color backColor = Color.White;
//variables
private int topX = 0, bottomX = 0;
int won = -1;
```

We need to instantiate the two panels used to house the UI components and the horses' animation. This is done using this constructor code:

```
ClientSize = new Size(500, 400);
SetupControlPanel();
drawingPanel = new Panel();
drawingPanel.Bounds = new Rectangle(controlPanel.Width, 0,
ClientSize.Width - controlPanel.Width, ClientSize.Height);
drawingPanel.BackColor = backColor;
Controls.Add(drawingPanel);
StartPosition = FormStartPosition.CenterScreen;
// THREAD CONTROL CODE GOES HERE
```

Note also a call to a function to set up all the UI components- **SetupControlPanel**. Inside your version of **SetupControlPanel** you must include the code to instantiate and set up the panels (this is new) and also the buttons (this is revision of properties and event handlers):

```
private void SetupControlPanel()
{
    controlPanel = new Panel();
    controlPanel.Bounds = new Rectangle(0, 0,
    ClientSize.Width / 5, ClientSize.Height);
    controlPanel.BackColor = Color.White;
    this.Controls.Add(controlPanel);
    // Buttons
    pauseButton = new Button();
    pauseButton.Text = "Start";
    pauseButton.Bounds = new Rectangle(8, 8, 85, 25);
    pauseButton.Click += new EventHandler(pauseButton_Click);
    controlPanel.Controls.Add(pauseButton);

    clearButton = new Button();
    clearButton.Text = "Clear";
    clearButton.Bounds = new Rectangle(8, pauseButton.Bounds.Y + pauseButton.Height +
5, 85, 25);
```

```
            clearButton.Click += new EventHandler(clearButton_Click);
            controlPanel.Controls.Add(clearButton);

            quitButton = new Button();
            quitButton.Text = "Quit";
            quitButton.Bounds = new Rectangle(8, clearButton.Bounds.Y + clearButton.Height +
5, 85, 25);
            quitButton.Click += new EventHandler(quitButton_Click);
            controlPanel.Controls.Add(quitButton);
        }

        private void clearButton_Click(object sender, EventArgs args)
        {
            Graphics g = drawingPanel.CreateGraphics();
            Brush brush = new SolidBrush(backColor);
            g.FillRectangle(brush, 0, 0, drawingPanel.Width, drawingPanel.Height);
            this.bottomX = 0;
            this.topX = 0;
            this.won = -1;
        }
```

Now we can set up the threading. We create a thread object and attach a function that will execute inside the thread independent of the main UI thread. This is one of the classic uses of threads, the UI and worker thread. In this case our horses are the workers and we want our UI to remain responsive as they race. The following line of code sets up the instances of the two threads (ALREADY included):

```
        private Thread DesertOrchid_t, RedRum_t;
```

The following code is then placed at the end of the **Animation** constructor (where indicated) which instantiates the threads and attaches the user-defined functions - in this case **RedRum** and **DesertOrchid**:

```
            ThreadStart drawStart_top = new ThreadStart(DesertOrchid);
            DesertOrchid_t = new Thread(drawStart_top);
            ThreadStart drawStart_bottom = new ThreadStart(RedRum);
            RedRum_t = new Thread(drawStart_bottom);
```

We now have two user-defined methods which will execute in parallel with the main form thread.

What we are looking for is a progression from one side of the screen to the other. This will be achieved by drawing a series of vertical lines and progressing a global x value (bottomX) by a set amount. Each thread will then be paused for a random amount of time before the next line is drawn. If we add to this some bounds checking for the global x value we can determine when the horse has reached the finish line. Once this has occurred we change the value of the semaphore variable won and change the colour of the Clear button to indicate who was victorious.

```
        /// <summary>
        /// Code for first horse
        /// </summary>
        private void RedRum()
        {
            Random random = new Random();
            Graphics g = drawingPanel.CreateGraphics();
            Pen pen = new Pen(Color.Black, 6);
            // Draw finish line
            g.DrawLine(pen, ClientSize.Width - 200, ClientSize.Height / 2, ClientSize.Width -
200, ClientSize.Height);

            while (true)
            {
                pen = new Pen(Color.FromArgb(random.Next(100) + 155, 100, 100), 4);
                if (this.topX > ClientSize.Width - 200 && this.won == -1)
                {
                    this.won = 1;
                    Console.WriteLine("RedRum won");
                    clearButton.BackColor = Color.FromArgb(random.Next(100) + 155, 100, 100);
                }
                g.DrawLine(pen, this.bottomX, ClientSize.Height / 2, this.bottomX += 4,
ClientSize.Height);

                Thread.Sleep(random.Next(100) + 40);
            }
        }

        /// <summary>
        /// Code for second horse
        /// </summary>
        private void DesertOrchid()
        {
            Random random = new Random();
            Graphics g = drawingPanel.CreateGraphics();
            Pen pen = new Pen(Color.Black, 6);
            // Draw finish line
            g.DrawLine(pen, ClientSize.Width - 200, ClientSize.Height / 2, ClientSize.Width -
200, ClientSize.Height);

            while (true)
            {
                pen = new Pen(Color.FromArgb(100, random.Next(100) + 155, 100), 4);
                if (this.topX > ClientSize.Width - 200 && this.won == -1)
                {
                    this.won = 1;
                    Console.WriteLine("Desert Orchid won");
                    clearButton.BackColor = Color.FromArgb(100, random.Next(100) + 155, 100);
                }
                g.DrawLine(pen, this.topX, 0, this.topX += 4, ClientSize.Height / 2);

                Thread.Sleep(random.Next(100) + 40);
            }
        }
```

We have buttons on our UI that can manipulate and change the state of our threads.We can interrogate a thread to find out its state {suspended | running | unstarted} and we can also change its state {thread.Resume() | thread.Start() | thread.Suspend()}.

```
        private void clearButton_Click(object sender, EventArgs args)
        {
            Graphics g = drawingPanel.CreateGraphics();
            Brush brush = new SolidBrush(backColor);
            g.FillRectangle(brush, 0, 0, drawingPanel.Width, drawingPanel.Height);
            this.bottomX = 0;
            this.topX = 0;
            this.won = -1;
            clearButton.BackColor = Color.White;
            Pen pen = new Pen(Color.Black, 6);
            g.DrawLine(pen, ClientSize.Width - 200, 1, ClientSize.Width - 200,
ClientSize.Height);
        }

        /// <summary>
        /// Kills the thread, then exits the application.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="args"></param>
        private void quitButton_Click(object sender, EventArgs args)
        {
```

```
    if ((DesertOrchid_t.ThreadState & ThreadState.Suspended) != 0)
        DesertOrchid_t.Resume();
    DesertOrchid_t.Abort();
    Application.Exit();
}

/// <summary>
/// Message handler for pauseButton. Starts the animation
/// the first time it is pressed. After that, it will toggle
/// between pausing and resuming the animation.
/// </summary>
/// <param name="sender"></param>
/// <param name="args"></param>
private void pauseButton_Click(object sender, EventArgs args)
{
    if (sender == pauseButton)
    {
        if ((DesertOrchid_t.ThreadState & ThreadState.Suspended) != 0)
        {
            DesertOrchid_t.Resume();
            pauseButton.Text = "Pause";
        }
        else if ((DesertOrchid_t.ThreadState & (ThreadState.Running |
        ThreadState.WaitSleepJoin)) != 0)
        {
            DesertOrchid_t.Suspend();
            pauseButton.Text = "Resume";
        }
        else if (DesertOrchid_t.ThreadState == ThreadState.Unstarted)
        {
            DesertOrchid_t.Start();
            pauseButton.Text = "Pause";
        }
    }
}
```

RUNNING THE C# CODE

Now run the code and you should see a representation of **Desert Orchid** "run" across the screen.

To create the race, extend the **Pause** button code (and **Quit** button code) to *also* change the state of the **Red Rum** thread. The winner is given by the colour of the **Clear** button, and is also printed to the console window.

Which horse would you expect to win the race? Run the race a dozen times – which horse wins each time? Explain this behaviour.

Change the **Console.WriteLine** instructions to **Messagebox.Show** instructions - what does this do to the behaviour of both threads?

What about three (or more) "horses"?

Happy racing!

END OF PRACTICAL