**,Developing a simple chat GUI application in C# (using threads & delegates)**

This tutorial looks at the development of a simple networked chat program using Visual Studio. As part of the tutorial, an introduction is made on the topic of using threads, as threads are utilised in the chat program. A example delegate is provided at the end of this tutorial.

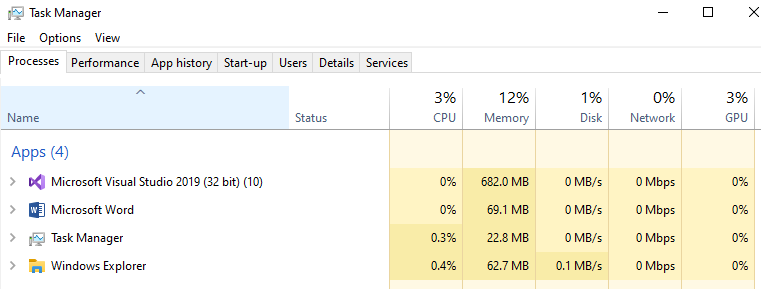
To follow this tutorial, you will need:

* Visual Studio (Community Edition 2022/2019 using C# .NET - required for Windows forms)
* Local web server (e.g. Apache) running localhost (127.0.0.1)

**An introduction to using threads**

On Windows, if you start the Task Manager (CTRL +ALT + DELETE and select Task Manager --- or --- RMB-click the bottom task bar to access), you will see a PROCESSES tab containing a listing of what processes are currently running.

The Windows Task Manager will display processes representing apps, background processes (running in the background without user intervention), and Windows-specific processes (running automatically as part of the Windows operating system).



*Screen shot of the app processes running on a PC using Windows 10 (Note: this may not be exactly what is running in your personal machine)*

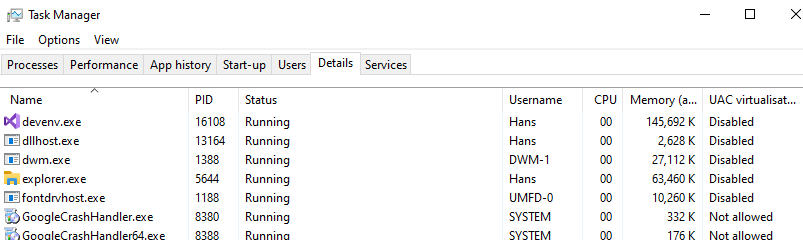
A **process** is an execution of a specific application program. For example, when you double-click to start the Google Chrome browser, you start a process that runs Google Chrome. When you open another instance of Google Chrome, you essentially create a second process.

A **thread** represents a sub-process, or a unit of execution being part of the host application process. So, a process can have one or more threads, all running concurrently - or to be more accurate - scheduled by the computer's operating system. Threads allow a process to operate more efficiently by doing multiple things simultaneously.

Whenever you create a process, a separate memory area is occupied; but threads share a common memory area. In C#, each thread has a life cycle. The life cycle of a thread is started when an instance of System.Threading.Thread class is created. When the task execution of the thread is completed, its life cycle is ended.

If you look at the DETAILS tab of the Task Manager, you will see a more detailed listing of the processes running on the computer.

Each process has its own unique process identifier (PID – Process Identifier) and its status (Running, Suspended) – refer screen shot example below.



**Why use threads?**

If an application is running as a process on Windows, the application can consist of one or more sub-processes – i.e. threads (more than one refers to multithreading). Multithreading allows the execution of multiple threads of a program to work simultaneously (or seemingly so). It is useful in environments where resources (e.g. memory, data, files, etc) are shared. For example, a web browser with multithreading can use one thread for user contact and another for image downloading at the same time.

**C# Thread Life Cycle**

In C#, each thread has a life cycle.   
The life cycle of a thread is started when instance of System.Threading.Thread class is created. When the task execution of the thread is completed, its life cycle is ended.

There are following states in the life cycle of a Thread in C#.

* Unstarted
* Runnable (Ready to run)
* Running
* Not Runnable
* Dead (Terminated)
* Unstarted State (when the instance of Thread class is created, it is in unstarted state by default)

Runnable State

When the Start() method on the thread is called, it is in runnable or ready to run state.

Running State

Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.

Not Runnable State

The thread is in not runnable state; if the Sleep()method is called on the thread, or input/output operation is blocked.

Dead State

After completing the task, thread enters into dead or terminated state.

**C# Thread class**

C# Thread class provides properties and methods to create and control threads. It is found in System.Threading namespace.

**Class Properties**

|  |  |
| --- | --- |
| **Property** | **Description** |
| CurrentThread | returns the instance of currently running thread |
| IsAlive | checks whether the current thread is alive or not - used to find the execution status of the thread |
| IsBackground | used to get or set value whether current thread is in background or not |
| ManagedThreadId | used to get unique id for the current managed thread |
| Name | used to get or set the name of the current thread |
| Priority | used to get or set the priority of the current thread |
| ThreadState | used to return a value representing the thread state |

**Class Methods**

|  |  |
| --- | --- |
| **Method** | **Purpose** |
| Abort() | used to terminate the thread --- it raises ThreadAbortException |
| Interrupt() | used to interrupt a thread which is in WaitSleepJoin state |
| Join() | used to block all the calling threads until this thread terminates |
| ResetAbort() | used to cancel the Abort request for the current thread |
| Sleep(Int32) | used to suspend the current thread for the specified milliseconds |
| Start() | changes the current state of the thread to Runnable |
| Yield() | used to yield the execution of current thread to another thread |

**Threads exercise**

Create a new project in Visual Studio and name it “ThreadExample”, or similar name.

Ensure that the following “using” statements are entered at the beginning. The System.Threading namespace contains the C# Thread class, which is used for this exercise.

using System;

using System.Threading;

Enter the following class inside the namespace and before class Program

namespace ThreadExample

{

// MyThread class - contains a simple Countdown()from 10 to 0

public class MyThread

{

public void Countdown ()

{

Thread t = Thread.CurrentThread;

Console.WriteLine(t.Name + " has started");

for (int i = 10; i >= 0; i--)

{

Console.WriteLine(t.Name + " - " + i);

// Sleep() method suspends the current thread

// for the specified milliseconds

// This is done to allow other threads

// get the chance to start execution

// Thread.Sleep(300);

}

Console.WriteLine(t.Name + " has ended");

}

}

Enter the following code in the Main() method

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Start of Main() method");

// Create instance of MyThread class

// NOTE: This is needed so that a non-static Countdown()  
 // method can run inside static Main() method

MyThread mt = new MyThread();

// Set up 3 thread objects to run separate countdown

Thread t1 = new Thread(new ThreadStart(mt.Countdown));

Thread t2 = new Thread(new ThreadStart(mt.Countdown));

Thread t3 = new Thread(new ThreadStart(mt.Countdown));

// Set up names for each thread

t1.Name = "Thread01";  
 // Join() causes all calling threads to wait until the  
 // the joined thread is terminated or completes its task

// t1.Join();

t2.Name = "Thread02";

t3.Name = "Thread03";

// Start() method is used to start the thread

t1.Start();

t2.Start();

t3.Start();

Console.WriteLine("End of Main() method");

} // End Main() method

} // End class Program

} // End namespace ThreadExample

Compile and run the program a few times. Note that the output is never the same. Why?

This is because the console application is its own process, running and executing code. The process is also host to 3 threads (separate sub-processes) all of which are running **asynchronously** with regard to one another. It is the operating system that arbitrarily decides (according to a well-defined priority policy), which thread (and process) gets a turn to execute its instructions.

One possible output of the console app with its 3 threads.

Start of Main() method  
End of Main() method  
Thread01 has started  
Thread01 – 10  
Thread01 – 9  
Thread01 – 8  
Thread01 – 7  
Thread01 – 6  
Thread01 – 5  
Thread01 – 4  
Thread01 – 3  
Thread01 – 2  
Thread01 – 1  
Thread01 – 0  
Thread01 has ended  
Thread03 has started  
Thread03 – 10  
Thread03 – 9  
Thread03 – 8  
Thread03 – 7  
Thread03 – 6  
Thread03 – 5  
Thread03 – 4  
Thread03 – 3  
Thread02 has started  
Thread02 – 10  
Thread02 – 9  
Thread02 – 8  
Thread02 – 7  
Thread02 – 6  
Thread02 – 5  
Thread02 – 4  
Thread02 – 3  
Thread02 – 2  
Thread02 – 1  
Thread03 – 2  
Thread03 – 1  
Thread03 – 0  
Thread03 has ended  
Thread02 – 0  
Thread02 has ended

Next, uncomment the Thread.Sleep() method call in the Countdown() method’s for loop

. The Sleep()method suspends the thread for the specified milliseconds (input parameter) in order to allow other threads get a chance to start and continue their execution.

for (int i = 10; i >= 0; i--)

{

Console.WriteLine(t.Name + " - " + i);

// Sleep() method suspends the current thread   
// for the specified milliseconds

// This is done to allow other threads

// get the chance to start execution

Thread.Sleep(300);

}

Run the program again, and note the difference in output. The Sleep() method is allowing time for the other threads to take turns in execution, as one sample output shows.

Start of Main() method  
End of Main() method  
Thread01 has started  
Thread01 – 10  
Thread03 has started  
Thread03 – 10  
Thread02 has started  
Thread02 – 10  
Thread01 – 9  
Thread02 – 9  
Thread03 – 9  
Thread01 – 8  
Thread02 – 8  
Thread03 – 8  
Thread01 – 7  
Thread03 – 7  
Thread02 – 7  
Thread01 – 6  
Thread03 – 6  
Thread02 - 6

---

Thread01 – 0  
Thread03 – 0  
Thread02 – 0  
Thread01 has ended  
Thread03 has ended  
Thread02 has ended

Run the program a few more times and note the arbitrary nature of the CPU resources in allocating time for the execution of each thread.

Next, uncomment the t1.Join() line of code in Main(). The Join() method will allow the calling thread to run its execution as priority over the other threads.

// Set up names for each thread

t1.Name = "Thread01";  
// Join() causes all the calling threads to wait until the  
// the joined thread is terminated or completes its task

t1.Join();  
t2.Name = "Thread02";

t3.Name = "Thread03";

Run the program and note that the entire execution of the t1 thread is given priority in full, despite the Sleep() method being called.

Typical output:

Start of Main() method  
Thread01 has started  
Thread01 – 10  
Thread01 – 9  
Thread01 – 8  
Thread01 – 7  
Thread01 – 6  
Thread01 – 5  
Thread01 – 4  
Thread01 – 3  
Thread01 – 2  
Thread01 – 1  
Thread01 – 0  
Thread01 has ended  
Thread02 has started  
Thread02 – 10  
End of Main() method  
Thread03 has started  
Thread03 – 10  
Thread03 – 9  
Thread02 – 9  
Thread03 – 8

---

Thread03 – 1  
Thread02 – 1  
Thread03 – 0  
Thread02 – 0  
Thread03 has ended  
Thread02 has ended

Re-comment the line t1.Join() so that it won’t execute.

Enter the following code just after the code that sets up the name for each thread in Main():

The Priority property is used to set a preferred priority level for each thread, however, the CPU resources will attempt to take this on board, but is not guaranteed.

// Set up names for each thread

t1.Name = "Thread01";

t2.Name = "Thread02";

t3.Name = "Thread03";

// Set priority levels for each thread

// NOTE: This is not guaranteed because thread execution

// is highly system-dependent

t3.Priority = ThreadPriority.Highest;

t2.Priority = ThreadPriority.Normal;

t1.Priority = ThreadPriority.Lowest;

Check the output to see if the CPU has observed the preferred priority specified above.  
Specifically, what order have the threads started and ended?  
Run the program several times. Is it consistent?  
How reliable is setting the Priority property given your tests?

Is the output consistently the same as the example below?

Thread03 has started  
Thread02 has started  
Thread01 has started

---

Thread03 has ended  
Thread02 has ended  
Thread01 has ended

Comment out the 3 lines of code that set each thread’s priority level.

Enter the following code just before the line of code which writes to the console “End of Main() method”:

The Abort() method terminates the thread and raises the ThreadAbortException.

try

{

// Abort() method is used to terminate the thread

t1.Abort();

t2.Abort();

t3.Abort();

}

catch (ThreadAbortException tae)

{

Console.WriteLine(tae.ToString());

}

Console.WriteLine(t1.Name + " status: " + t1.IsAlive);

Console.WriteLine(t2.Name + " status: " + t2.IsAlive);

Console.WriteLine(t3.Name + " status: " + t3.IsAlive);

Console.WriteLine("End of Main() method");

Run the program and check the result (sample output below, but may be different to your own result).

Start of Main() method  
Thread01 has started  
Thread02 has started  
Thread02 – 10  
Thread03 has started  
Thread03 – 10  
Thread01 status: False  
Thread02 status: False  
Thread03 status: False  
End of Main() method

Comment out the try/catch block and the 3 lines which output the IsAlive property for each thread.

**Synchronization** is a technique that allows only one thread to access a resource for the particular time. No other thread can interrupt until the assigned thread finishes its task.

In a multithreading program, threads are allowed to access any resource for the required execution time. Threads essentially share resources and execute asynchronously.

Accessing shared resources (data) is critical task that sometimes may halt the system. To avoid these potential issues, threads are often made synchronized. If a program dealt with transactions like deposits and withdrawals, consistency and accuracy is vital and there should be no interference between threads.

C# uses the 'lock' keyword to execute a thread synchronously within a program. It is used to get lock for the current thread, execute the task and then release the lock. This ensures that any other thread does not interrupt the execution until the execution is completed.

Check what a ‘deadlock’ is here:  
<https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/>

In the Countdown() method of the MyThread class, enter the line of code which sets up the synchronization of the thread … lock (this); --- and comment out the Thread.Sleep(300); line … as per the example below:

public void Countdown ()

{

// Synchronization

lock (this);

Thread t = Thread.CurrentThread;

Console.WriteLine(t.Name + " has started");

for (int i = 10; i >= 0; i--)

{

Console.WriteLine(t.Name + " - " + i);

// Sleep() method suspends the current thread

// Thread.Sleep(300);

Console.WriteLine(t.Name + " has ended");

}

Run the program a few times and you should see that each thread executes its own Countdown() method with a completed for loop --- most of the time.

Example output:

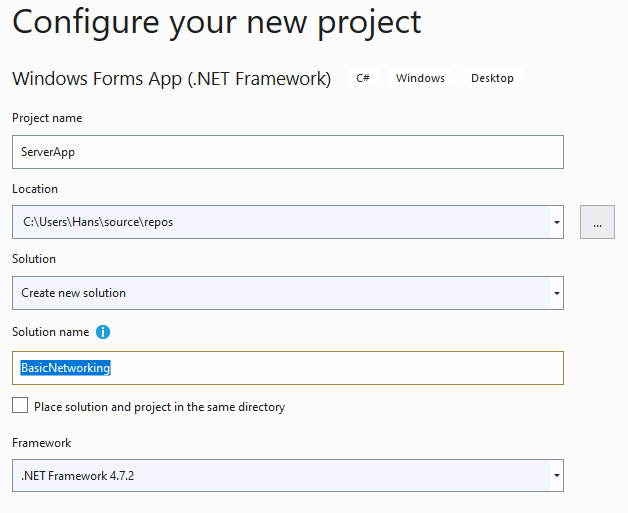
Start of Main() method  
Thread01 has started  
End of Main() method  
Thread03 has started  
Thread03 – 10  
Thread03 – 9  
Thread03 – 8  
Thread03 – 7  
Thread03 – 6  
Thread03 – 5  
Thread03 – 4  
Thread03 – 3  
Thread03 – 2  
Thread03 – 1  
Thread03 – 0  
Thread03 has ended  
Thread01 – 10  
Thread01 – 9  
Thread01 – 8  
Thread01 – 7  
Thread01 – 6  
Thread01 – 5  
Thread01 – 4  
Thread01 – 3  
Thread01 – 2  
Thread01 – 1  
Thread01 – 0  
Thread01 has ended  
Thread02 has started  
Thread02 – 10  
Thread02 – 9  
Thread02 – 8  
Thread02 – 7  
Thread02 – 6  
Thread02 – 5  
Thread02 – 4  
Thread02 – 3  
Thread02 – 2  
Thread02 – 1  
Thread02 – 0  
Thread02 has ended

Uncomment the Sleep() method this time, and run the program again. What is the output? How different is it to when the Sleep() method was not used. What is the reason for the difference?  
Remember that Sleep() is used to **suspend** the current thread in favour of allowing time for other threads to execute, despite the lock.

Start of Main() method  
End of Main() method  
Thread01 has started  
Thread02 has started  
Thread02 – 10  
Thread01 – 10  
Thread03 has started  
Thread03 – 10  
Thread02 – 9  
Thread03 – 9  
Thread01 – 9  
Thread03 – 8  
Thread02 – 8  
Thread01 – 8  
Thread03 – 7  
Thread01 – 7  
Thread02 – 7  
Thread02 – 6  
Thread01 – 6  
Thread03 – 6  
Thread02 – 5  
Thread01 – 5  
Thread03 – 5  
Thread01 – 4  
Thread02 – 4  
Thread03 – 4  
Thread02 – 3  
Thread01 – 3  
Thread03 – 3  
Thread02 – 2  
Thread01 – 2  
Thread03 – 2  
Thread02 – 1  
Thread01 – 1  
Thread03 – 1  
Thread02 – 0  
Thread01 – 0  
Thread03 – 0  
Thread02 has ended  
Thread01 has ended  
Thread03 has ended

**Basic networking between two form applications (server and client)**

Create a new Visual C# project in Visual Studio using Windows Forms App (.NET Framework).  
Name the project “ServerApp” or alternative suitable name and name the solution as “BasicNetworking”.  
Select an appropriate location and click the FINISH button to accept.



A Windows Form should appear.

Go to the properties and change the following:

* Name: “ServerForm”
* Size: 600, 400
* Text: “Server”

Using the toolbox of form controls, set up the following:

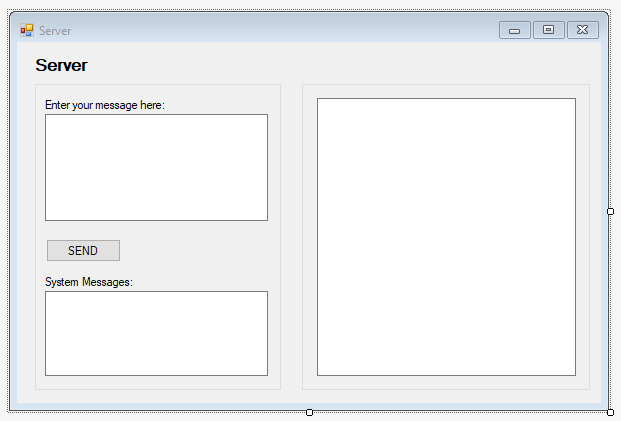
1 x Label (for title --- enter “Server” as the text title)  
Make the FONT > SIZE = 12 and FONT > BOLD = true.

GroupBox (on left) containing

* 2 x Labels (one set up with text “Enter your message here:” and the other with “System messages:”)
* 2 x multi-line TextBoxes (top box is for messages to send and bottom box is for error messages)  
  Name the top text box “Send\_TextBox”  
  Name the bottom text box “SystemMsg\_TextBox”
* 1 x Button (used to send the message)  
  Name the button “Send\_Button” and use “SEND” as the text on the button.

GroupBox (on right) containing:

* 1 x multi-line TextBox (accumulated messages sent and received during a chat session)  
  Name the box “Receive\_TextBox”



Example Server form in design view.

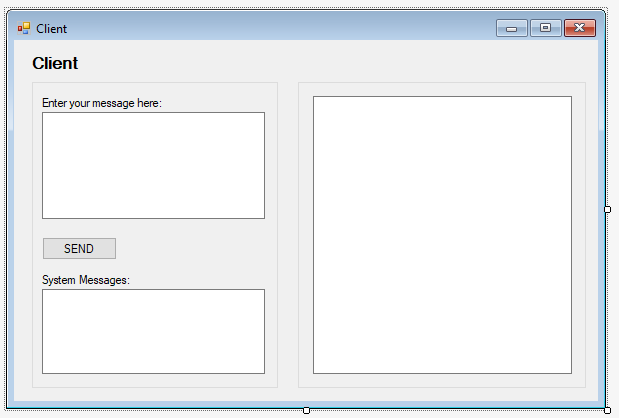
RMB-click the solution and select ADD > NEW PROJECT (this will be the Client form application which will be part of the existing solution named “BasicNetworking”).

Name the project “ClientApp”.

Go to the properties and change the following:

* Name: “ClientForm”
* Size: 600, 400
* Text: “Client”

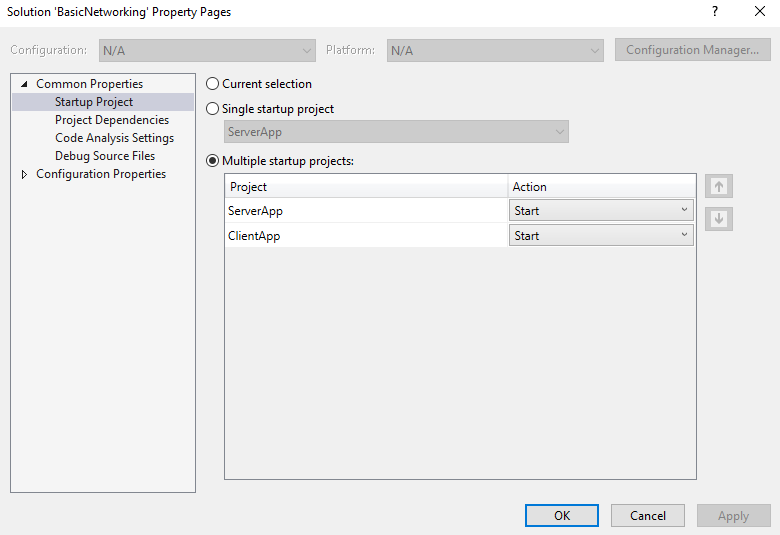
Copy the same controls set up for the Server for the Client application.  
You can do this by marquee-selecting the controls in the Server and using COPY + PASTE into the Client application. This will also set up the same control names for the duplicates.



Example Client application in design view

RMB-click the Solution and choose PROPERTIES.

Choose the Multiple Startup projects (select radio button), and set both ServerForm and ClientForm to “Start”. Ensure the ServerApp is listed first in order of the two applications (refer example screen shot below). Click the OK button to accept.



This will display <Multiple Startup Projects> in the solution shelf.  
Next, click the START button (green arrow) to launch both form applications.



You should see both form applications running.

Open the SOURCE CODE for the **ServerApp Form**.

Ensure the following using statements are entered:

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;

using System.Windows.Forms;

Enter the following data fields inside the ServerApp class (before the constructor method)  
  
public bool exitStatus = false;

public const int BYTE\_SIZE = 1024;

public const int PORT\_NUMBER = 8888;

// listens for and accept incoming connection requests

private TcpListener serverListener;

// TcpClient is used to connect with the TcpListener object

private TcpClient serverSocket;

// set up data stream object

private NetworkStream netStream;

// set up thread to run ReceiveStream() method

private Thread serverThread = null;

// set up delegate

// a delegate is a reference variable to a method

// and used for a call back by the delegate object

// delegate ref variable is declared in SetText() method below

delegate void SetTextCallback (string text);

Inside the public ServerApp()constructor, enter the following code (after InitializeComponent():

public ServerApp()

{

InitializeComponent();

// clear all text boxes

SystemMsg\_TextBox.Text = "Error messages appear here ...";

Send\_TextBox.Text = "Enter text here and press send button ...";

Receive\_TextBox.Text = "";

// run server

StartServer();

}

Below the constructor method, create the StartServer()method as follows:

private void StartServer()

{

try

{

// create listener and start

serverListener = new TcpListener(IPAddress.Loopback, PORT\_NUMBER);

serverListener.Start();

// create acceptance socket

// this creates a socket connection for the server

serverSocket = serverListener.AcceptTcpClient();

// create stream

netStream = serverSocket.GetStream();

// set up thread to run ReceiveStream() method

serverThread = new Thread(ReceiveStream);

// start thread

serverThread.Start();

Receive\_TextBox.Text = "Server started ..." + Environment.NewLine;

}

catch (Exception e)

{

// display exception message

SystemMsg\_TextBox.Text = e.StackTrace;

}

}

Next, create the ReceiveStream()method:

// this method runs as a thread (called by serverThread)

public void ReceiveStream()

{

byte[] bytesReceived = new byte[BYTE\_SIZE];

// loop to read any incoming messages

while (! exitStatus)

{

try

{

int bytesRead = netStream.Read(bytesReceived, 0, bytesReceived.Length);

this.SetText(Encoding.ASCII.GetString(bytesReceived, 0, bytesRead));

}

catch (System.IO.IOException)

{

Console.WriteLine("Client has exited!");

exitStatus = true;

}

}

}

Next, create the SetText() method:

private void SetText(string text)

{

// InvokeRequired compares the thread ID of the

// calling thread to the thread ID of the creating thread.

// if these threads are different, it returns true.

if (this.Receive\_TextBox.InvokeRequired)

{

// d is a Delegate reference to the SetText() method

SetTextCallback d = new SetTextCallback(SetText);

this.Invoke(d, new object[] { text });

}

else

{

this.Receive\_TextBox.Text += "Client: " + text + Environment.NewLine;

}

}

In the Server Form (DESIGN VIEW), LMB-double click the SEND button control, and enter the following code for the event handler created:

private void Send\_Button\_Click(object sender, EventArgs e)

{

// send message in Send\_TextBox if any text present

if (Send\_TextBox.Text.Length > 0)

{

// construct byte array to stream in write mode

String strToSend = Send\_TextBox.Text;

byte[] bytesToSend = Encoding.ASCII.GetBytes(strToSend);

netStream.Write(bytesToSend, 0, bytesToSend.Length);

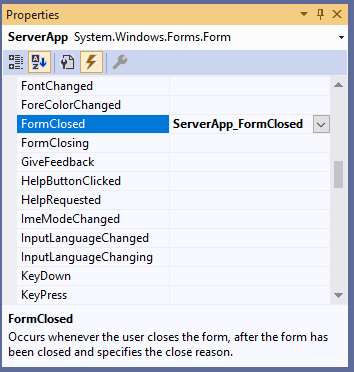
Receive\_TextBox.Text += "Server: " + strToSend + Environment.NewLine;

Send\_TextBox.Text = "";

}

}

LMB-click the top of the ServerApp Form application in DESIGN VIEW and in the PROPERTIES (EVENTS), select FORM CLOSED --- LMB-double click to create an event handler (automatically names as “ServerApp\_FormClosed()”.



Enter the following code for this event handler:

private void ServerApp\_FormClosed(object sender, FormClosedEventArgs e)

{

// terminate thread if still running

if (serverThread.IsAlive)

{

Console.WriteLine("Server thread is alive");

serverThread.Interrupt();

if (serverThread.IsAlive)

{

Console.WriteLine("Server thread is now terminated");

}

}

else

{

Console.WriteLine("Server thread is terminated");

}

// close the application for good

Environment.Exit(0);

}

This finalises the code for the Server form.

Open the SOURCE CODE for the **ClientApp Form**.

Enter the following data fields inside the ClientApp class (just before the constructor method).

public bool exitStatus = false;

public const int BYTE\_SIZE = 1024;

public const string HOST\_NAME = "localhost";

public const int PORT\_NUMBER = 8888;

// set up a client connection for TCP network service

private TcpClient clientSocket;

// set up data stream object

private NetworkStream netStream;

// set up thread to run ReceiveStream() method

private Thread clientThread = null;

// set up delegate

delegate void SetTextCallback (string text);

Inside the public ClientApp()constructor, enter the following code (after InitializeComponent()):

public ClientApp()

{

InitializeComponent();

// clear all text boxes

SystemMsg\_TextBox.Text = "Error messages appear here ...";

Send\_TextBox.Text = "Enter text here and press send button ...";

Receive\_TextBox.Text = "";

// start client

StartClient();

}

Below the constructor method, create the StartClient()method as follows:

private void StartClient()

{

try

{

// create TCPClient object (as the socket)

clientSocket = new TcpClient(HOST\_NAME, PORT\_NUMBER);

// create stream

netStream = clientSocket.GetStream();

// set up thread to run ReceiveStream() method

clientThread = new Thread(ReceiveStream);

// start thread

clientThread.Start();

Receive\_TextBox.Text = "Client started ..." + Environment.NewLine;

}

catch (Exception e)

{

// display exception message

SystemMsg\_TextBox.Text = e.StackTrace;

}

}

Next, create the ReceiveStream()method:

// this method runs as a thread (called by serverThread)

public void ReceiveStream()

{

byte[] bytesReceived = new byte[BYTE\_SIZE];

// loop to read any incoming messages

while (! exitStatus)

{

try

{

int bytesRead = netStream.Read(bytesReceived, 0, bytesReceived.Length);

this.SetText(Encoding.ASCII.GetString(bytesReceived, 0, bytesRead));

}

catch (System.IO.IOException)

{

Console.WriteLine("Server has exited!");

exitStatus = true;

}

}

}

Next, create the SetText() method:

private void SetText(string text)

{

// InvokeRequired compares the thread ID of the

// calling thread to the thread ID of the creating thread.

// if these threads are different, it returns true.

if (this.Receive\_TextBox.InvokeRequired)

{

// d is a Delegate reference to the SetText() method

SetTextCallback d = new SetTextCallback(SetText);

this.Invoke(d, new object[] { text });

}

else

{

this.Receive\_TextBox.Text += "Server: " + text + Environment.NewLine;

}

}

In the Client Form (DESIGN VIEW), LMB-double click the SEND button control, and enter the following code for the event handler created:

private void Send\_Button\_Click(object sender, EventArgs e)

{

// send message in Send\_TextBox if any text present

if (Send\_TextBox.Text.Length > 0)

{

// construct byte array to stream in write mode

String strToSend = Send\_TextBox.Text;

byte[] bytesToSend = Encoding.ASCII.GetBytes(strToSend);

netStream.Write(bytesToSend, 0, bytesToSend.Length);

Receive\_TextBox.Text += "Client: " + strToSend + Environment.NewLine;

Send\_TextBox.Text = "";

}

}

As per the ServerApp Form (i.e. the FormClosed event), create a similar event handler for the ClientApp Form. This will be for the FORM CLOSED event.

Enter the following code for this event handler:

private void ClientApp\_FormClosed(object sender, FormClosedEventArgs e)

{

// terminate client thread if still running

if (clientThread.IsAlive)

{

Console.WriteLine("Client thread is alive");

clientThread.Interrupt();

if (clientThread.IsAlive)

{

Console.WriteLine("Client thread is now terminated");

}

}

else

{

Console.WriteLine("Client thread is terminated");

}

// close the application for good

Environment.Exit(0);

}

This finalises the code for the ClientApp form.

**Note:**

This basic chat example using two form applications only uses localhost (IP address: 127.0.0.1) and an arbitrarily assigned port number of 8888.

The provided code does not allow multiple clients or any form of security.

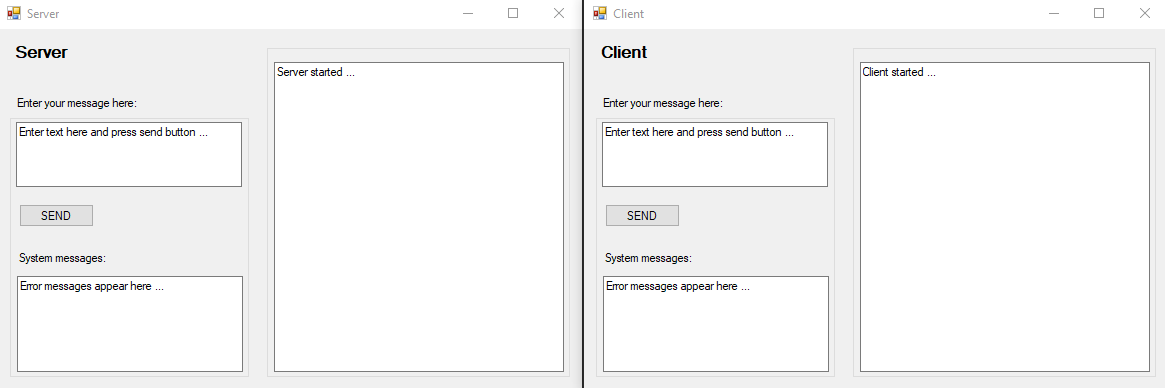
**Test run the applications**

Start the applications by clicked the START button (with green arrow)

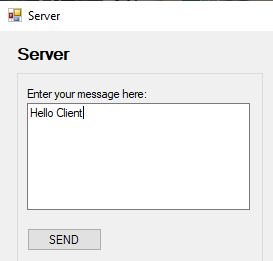


Alternatively, you may use DEBUG > START WITHOUT DEBUGGING or use CTRL + F5 buttons.

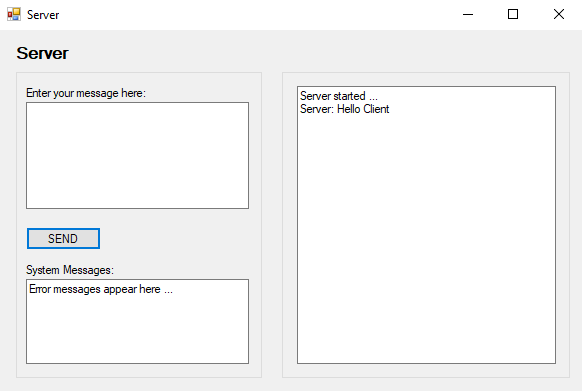
Place the two applications side by side.



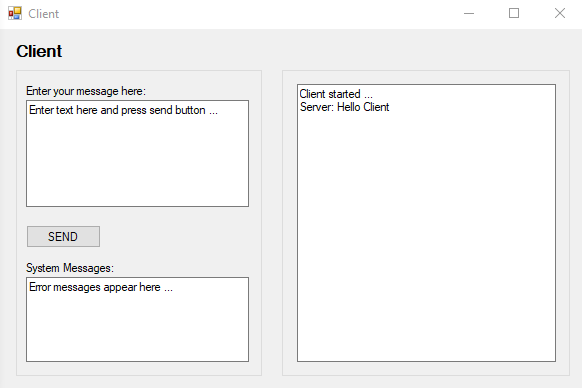
In the SERVER app, enter the message “Hello Client” in the send text field (top left), and press the SEND button.



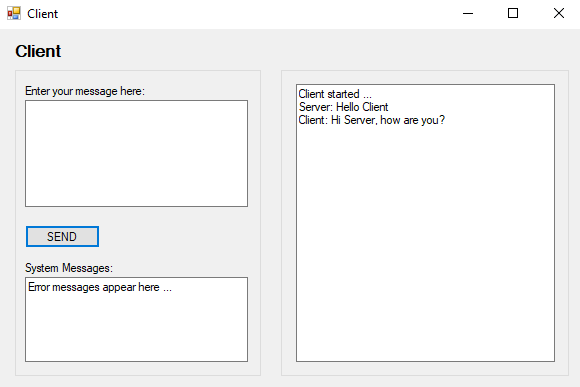
After clicking the SEND button in the SERVER app, the message should appear in the larger text area on the right (named “Receive\_TextBox”).  
The text box where the message was entered should be cleared (named “Send\_TextBox”).

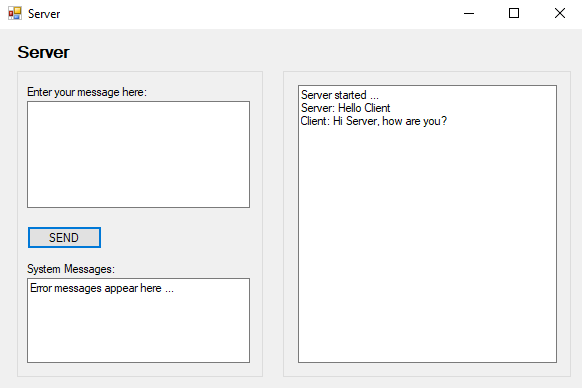


The CLIENT app should display the sent message in Receive\_TextBox as well.



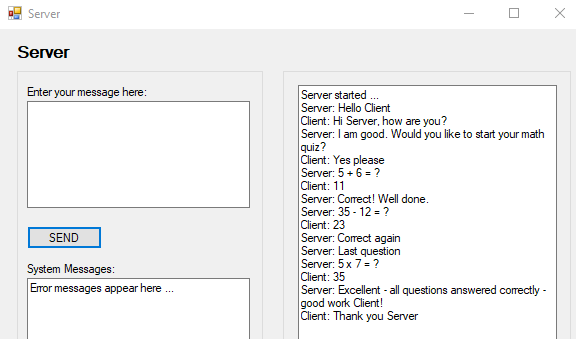
Use the CLIENT app to respond.  
Enter the message “Hi Server, how are you?” and press the SEND button (in the CLIENT app).  
You should see the client message appear in each app.

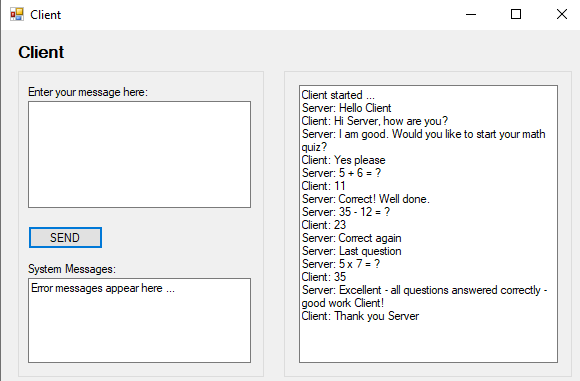




Test the chat applications further, by making the SERVER send a math question and the CLIENT responding with an answer.

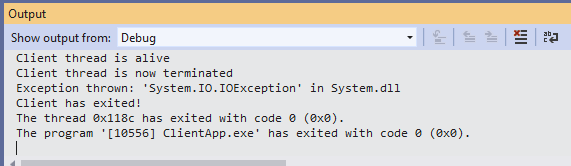
Example screen shots of chat with 3 math questions between SERVER and CLIENT.



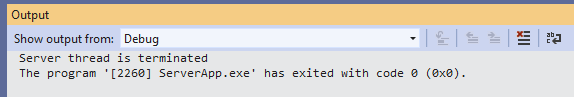


To end the chat session, simple click the WINDOW EXIT ‘X’ icon in either application.

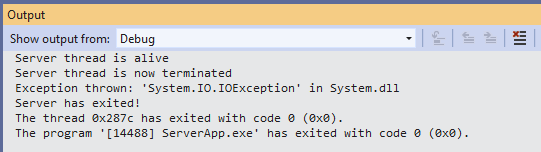
If the CLIENT application is first to shut down, the following should be displayed in the OUTPUT panel.



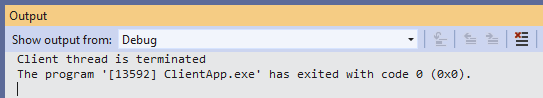
And when the SERVER application is next to shut down, the OUTPUT displays the fact that the Server thread is terminated and the program ServerApp.exe has exited.



If the SERVER application is first to shut down, the following is displayed in the OUTPUT panel.



And when the CLIENT application follows, the OUTPUT displays the fact that the Client thread is terminated and the program ClientApp.exe has exited.



Note:

In both cases, a System.IO.IOException is thrown from within the while loop of the ReceiveStream() method set up in each application.

In the case of the CLIENT application being shut down, the SERVER is still running the while loop using the condition that checks its exitStatus boolean.

The try block code attempts to use the netStream.Read() but there is no CLIENT. This creates the IOException which is caught in the catch block.

The catch block outputs that the CLIENT has exited and sets the exitStatus boolean to true and exits the while loop.

**Networking basics**

A **socket** is one end point of a two-way communication link between two applications running on a network. A socket is bound to a port number so that the TCP layer can identify the application that the data is destined to be sent to.

**TCP or Transmission Control Protocol** is a standard that defines how to establish and maintain a network connection through which application programs that can exchange data. TCP works with IP or Internet Protocol, which defines how computers can send packets of data to each other.

The **TcpListener** C# .NET class implements methods that listen for and accept incoming connection requests. To set up an instance, an IP address and a port number are required. The SERVER application only utilises this instance.

The **TcpClient** C# class implements methods that connect, send and receive stream data over a network. Instances for this class are used in both the SERVER and CLIENT applications.

The **NetworkStream** C# class provides methods for sending and receiving data. To create a NetworkStream object, you must provide a connected socket object. The two methods utilised in this simple chat example are Read() and Write().

**Useful Links (networking basics):**

Stackoverflow - What is a Socket?  
<https://stackoverflow.com/questions/16233193/what-exactly-is-socket>

TcpListener class  
<https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.tcplistener?view=netcore-3.1>

TcpClient class  
<https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.tcpclient?view=netcore-3.1>

NetworkSteam class  
<https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.networkstream?view=netcore-3.1>

**TCP/IP Ports and Protocols**

A **port** is an end point to a logical connection and the means a client program specifies a particular server program (with its own IP address) in a network.

The **port number** identifies what type of port it is.

**Well-known ports** are port numbers that are reserved for privileged services. Well-known ports range from 0 through 1023. The following table represents the most common port numbers:

|  |  |
| --- | --- |
| Port Number | Assigned Protocol |
| 20 | FTP - Data (File Transfer Protocol) |
| 21 | FTP - Control |
| 22 | SSH (Secure Shell) |
| 23 | Telnet |
| 25 | SMTP (Simple Mail Transfer Protocol) |
| 53 | DNS (Domain Name System) |
| 80 | HTTP (Hypertext Transfer Protocol) |

**Registered Ports** are 1024 to 49151 and **Dynamic Ports** (also called private ports) are 49152 to 65535.

For more information, refer RFC (Request for Comments) 1700:  
<https://www.ietf.org/rfc/rfc1700.txt?number=1700>

**More about Threads**

A thread object has been set up for each SERVER and CLIENT applications.

A thread is an execution context, which is all the information a CPU needs to execute a stream of instructions.

Suppose you're reading a book, and you want to take a break right now, but you want to be able to come back and resume reading from the exact point where you stopped. One way to achieve that is by jotting down the page number, line number, and word number. So your execution context for reading a book is these 3 numbers.

If you have a roommate, and she's using the same technique, she can take the book while you're not using it, and resume reading from where she stopped. Then you can take it back, and resume it from where you were.

Threads work in the same way. A CPU is giving you the illusion that it's doing multiple computations at the same time. It does that by spending a bit of time on each computation. It can do that because it has an execution context for each computation. Just like you can share a book with your friend, many tasks can share a CPU.

On a more technical level, an execution context (therefore a thread) consists of the values of the CPU's registers.

Threads are different from processes. A thread is a context of execution, while a process is a bunch of resources associated with a computation. A process can have one or many threads.

The resources associated with a process include memory pages (all the threads in a process have the same view of the memory), file descriptors (e.g., open sockets), and security credentials (e.g., the ID of the user who started the process).

From Wikipedia (<https://en.wikipedia.org/wiki/Thread_(computing)>)  
*In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its dynamically allocated variables and non-thread-local global variables at any given time.*

**Commonly used Thread methods**

* **Start()** Starts a thread
* **Sleep(n)** Suspends a thread for n milliseconds, then resumes
* **Join()** Blocks the calling thread until it terminates
* **Abort()** Raises a ThreadAbortException in the thread to begin the termination process
* **Interrupt()** Interrupts a thread in the WaitSleepJoin thread state

**Important Points about using Threads:**

* A deadlock can occur if the thread that calls Abort methods holds a lock that the aborted thread requires.
* If the Abort method is called on a thread which has not been started, then that thread will abort when Start is called.
* If the Abort method is called on a thread which is blocked or is sleeping then the thread will get interrupted and after that get aborted.
* If Abort method is called on a suspended thread then a ThreadStateException is thrown in the thread that called Abort, and AbortRequested is added to the ThreadState property of the thread being aborted.
* A ThreadAbortException is not thrown in the suspended thread until Resume is called.
* If the Abort method is called on a Managed thread which is currently executing unmanaged code then a ThreadAbortException is not thrown until the thread returns to managed code.
* If two calls to Abort come at the same time then it is possible for one call to set the state information and the other call to execute the Abort. But, an application cannot detect this situation.
* After Abort is called on a thread, the state of the thread includes AbortRequested. After the thread has terminated due to the result of a successful call to Abort, the state of the thread is changed to Stopped. With sufficient permissions, a thread which is the target of an Abort can cancel the abort using the ResetAbort method.

**Useful Links (threads):**

Microsoft .NET Documentation  
Thread class  
<https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=netcore-3.1>

Stackoverflow - What is a Thread?  
<https://stackoverflow.com/questions/5201852/what-is-a-thread-really>

**More about C# Delegates**

A delegate in C# is a reference variable to a method.

When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Refer Link:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/>

Delegates have the following properties:

- Delegates are similar to C++ function pointers, but are type safe.

- Delegates allow methods to be passed as parameters.

- Delegates can be used to define callback methods.

- Delegates can be chained together; for example, multiple methods can be called on a single event.

- Methods don't need to match the delegate signature exactly.

- Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code that can call the referenced method, without having to know at compile time which method will be invoked.

An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

A delegate has 3 steps

- Declaration

- Instantiation

- Invocation

For the chat example, a delegate is used for a call back by the delegate object – the delegate reference variable is declared in the SetText() method. A "callback" is a term that refers to a coding design pattern. In this design pattern, executable code is passed as an argument to other code and it is expected to call back at some time. This callback can be synchronous or asynchronous.

**Useful Links (C# delegates):**

Microsoft .NET Documentation  
<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/>

C Sharp corner  
<https://www.c-sharpcorner.com/UploadFile/1c8574/delegate-used-for-callback-operation/>

Tutorials Point  
<https://www.tutorialspoint.com/csharp/csharp_delegates.htm>

The following C# code is an interesting Delegate example that combines string arrays, sorts and removes duplicates.

using System;

using System.Collections.Generic;

namespace DelegateExample

{

internal class Program

{

// create 2 delegates

delegate string [] Combine (string [] strArray1, string [] strArray2);

delegate string[] ArrayOperation(string[] strArray);

static void Main(string[] args)

{

// create 3 string arrays of actor names from Star Wars films

string[] namesArray1 = { "Mark Hamill", "Harrison Ford", "Carrie Fisher", "Alec Guiness", "James Earl Jones", "Phil Brown"};

string[] namesArray2 = { "Mark Hamill", "Harrison Ford", "Carrie Fisher", "Alec Guiness", "James Earl Jones", "Frank Oz"};

string[] namesArray3 = { "Mark Hamill", "Harrison Ford", "Carrie Fisher", "James Earl Jones", "Frank Oz", "Billy Dee Williams", "Tim Rose" };

// declare delegate variable to combine string arrays

Combine combine = new Combine (CombineArrays);

string[] combinedArray = combine(combine(namesArray1, namesArray2), namesArray3);

// display combinedArray

Console.WriteLine("Combined Array (unsorted)...");

DisplayArray(combinedArray);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

// declare delegate variable to sort a string array

ArrayOperation arrayOp = new ArrayOperation(SortArray);

string[] sortedCombinedArray = arrayOp(combinedArray);

// display sorted combinedArray

Console.WriteLine("Combined Array (sorted)...");

DisplayArray(sortedCombinedArray);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

// reassign delegate variable to remove duplicates  
 // note: the RemoveDuplicates() method has the same output

// and the same input as the SortArray() method  
 // so we can use the same delegate variable

arrayOp = new ArrayOperation(RemoveDuplicates);

string[] noDuplicatesSortedCombinedArray = arrayOp (sortedCombinedArray);

// display sorted combinedArray with no duplicates

Console.WriteLine("Combined Array (sorted with no duplicates)...");

DisplayArray(noDuplicatesSortedCombinedArray);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

// CombineArrays() method (combines two arrays and returns the one array)

public static string [] CombineArrays (string [] array1, string [] array2)

{

string[] combinedArray;

if (array1.Length > 0 && array2.Length > 0)

{

combinedArray = new string[array1.Length + array2.Length];

Array.Copy (array1, 0, combinedArray, 0, array1.Length);

Array.Copy(array2, 0, combinedArray, array1.Length, array2.Length);

}

else if (array1.Length > 0)

{

combinedArray = new string[array1.Length];

Array.Copy(array1, 0, combinedArray, 0, array1.Length);

}

else if (array2.Length > 0)

{

combinedArray = new string[array2.Length];

Array.Copy(array2, 0, combinedArray, 0, array2.Length);

}

else

{

combinedArray = new string[1];

combinedArray[0] = "";

}

return combinedArray;

}

// SortArray() method sorts the input string array using List Sort() method

public static string [] SortArray (string [] array)

{

// List of string values from string [] array input

List<string> strList = new List<string>(array);

// Alphabetically sort the list

strList.Sort();

// Return sorted array

return strList.ToArray ();

}

// RemoveDuplicates() method removes duplicates from input string array

public static string[] RemoveDuplicates(string[] array)

{

List<string> strList = new List<string>(array);

List<string> strListNoDuplicates = new List<string> ();

strListNoDuplicates.Add(strList[0]);

for (int i = 1; i < strList.Count; i++)

{

string str = strList[i];

bool duplicateStatus = false;

for (int j = 0; j < strListNoDuplicates.Count; j++)

{

if (str.Equals(strListNoDuplicates[j]))

{

duplicateStatus = true;

break;

}

}

if (!duplicateStatus)

{

strListNoDuplicates.Add(str);

}

}

string[] uniqueStr = strListNoDuplicates.ToArray();

return uniqueStr;

}

public static void DisplayArray (string [] array)

{

foreach (string str in array)

{

Console.WriteLine (str);

}

Console.WriteLine ();

}

}

}

/\*

Sample output:

Combined Array (unsorted)...

Mark Hamill

Harrison Ford

Carrie Fisher

Alec Guiness

James Earl Jones

Phil Brown

Mark Hamill

Harrison Ford

Carrie Fisher

Alec Guiness

James Earl Jones

Frank Oz

Mark Hamill

Harrison Ford

Carrie Fisher

James Earl Jones

Frank Oz

Billy Dee Williams

Tim Rose

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Combined Array (sorted)...

Alec Guiness

Alec Guiness

Billy Dee Williams

Carrie Fisher

Carrie Fisher

Carrie Fisher

Frank Oz

Frank Oz

Harrison Ford

Harrison Ford

Harrison Ford

James Earl Jones

James Earl Jones

James Earl Jones

Mark Hamill

Mark Hamill

Mark Hamill

Phil Brown

Tim Rose

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Combined Array (sorted with no duplicates)...

Alec Guiness

Billy Dee Williams

Carrie Fisher

Frank Oz

Harrison Ford

James Earl Jones

Mark Hamill

Phil Brown

Tim Rose

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/