**Task 1 (MultiThreading.Task1.100Tasks.csproj):**

Open *100Task*s project and write a program that creates an array of 100 Tasks, runs them and waits till all of them are completed. Each Task should iterate from 1 to 1000 and print to the console the following string: *“Task #0 – {iteration number}”.*

**Task 2 (MultiThreading.Task2.Chaining.csproj):**

Open *Chaining* project and write a program that creates a chain of four Tasks:

* 1st Task creates an array of 10 random integers.
* 2nd Task multiplies this array with another random integer.
* 3rd Task sorts this array by ascending.
* 4th Task calculates the average value.

All these tasks should print the values to console.

**Task 3 (MultiThreading.Task3.MatrixMultiplier.csproj, MultiThreading.Task3.MatrixMultiplier.Tests.csproj)**:

Open *MatrixMultiplier* project and write a program that multiplies two matrices following the conditions below:

* Implement the logic of *MatricesMultiplierParallel* class using Parallel class. All unit tests within *MatrixMultiplier.Tests project* should pass successfully.
* Create a test inside MatrixMultiplier.Tests project to check which of the multiplier implementations (synchronous or parallel) runs faster. Find out the size that makes parallel multiplication more effective than the regular one.

**Task 4 (MultiThreading.Task4.Threads.Join.csproj):**

Open *Threads.Join* project to create a program that recursively creates 10 threads. Each thread should be with the same body and receive a state with an integer number, decrement it, print and pass as a state into the newly created thread. Implement all the following options:

* Use Thread class for this task and Join for waiting threads.
* Use ThreadPool class for this task and Semaphore for waiting threads.

**Task 5 (MultiThreading.Task5.Threads.SharedCollection.csproj):**

Open *Threads.SharedCollection* and write a program that creates two threads and a shared collection:

* The 1st task should add 10 elements to the collection
* The 2nd task should print all elements in the collection after each adding (in other words, if the collection contains elements from 1 to 10, the second thread should print something like [1], [1, 2], [1, 2, 3], …, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], the number of elements should increase).

Use Thread, ThreadPool or Task classes for thread creation and any kind of synchronization constructions.

**Task 6 (MultiThreading.Task6.Continuation.csproj):**

In the *Continuation* project you need to create a task which should be attached to the parent with the following continuation criteria:

* Continuation task should be executed regardless of the result of the parent task.
* Continuation task should be executed when the parent task was completed without success.
* Continuation task should be executed when the parent task failed and parent task thread should be reused for continuation
* Continuation task should be executed outside of the thread pool when the parent task is cancelled

Demonstrate the work of each case with console utility.

**Home task (Optional):**

Our task is to create a client and a server for simple enterprise chat.

**General requirements:**

* Both Client and Server should be implemented as Console or GUI applications (you decide)
* Client and Server interacts with each other using Named Pipes (System.IO.Pipes) or Sockets (System.Net.Sockets) – you decide. For simplicity, connection parameters could be hardcoded.
* Client is a bot which performs the following operations in a loop:
  + Connects to the Server with a new name
  + Sends several messages to the Server with a short delay between messages. (Messages are retrieved from the list of already predefined messages, the number of messages and delay between messages is random)
  + Receives all messages from the Server and displays them on the screen or stores in a text file.
  + Disconnects from the Server.

Repeat the loop until the User stops the Client or exception occurred.

**Server:**

* Accepts connections from the Client. On connecting receives the name of the Client.
* Receives messages from the Client and broadcasts them to all other clients connected to this Server .
* Stores a history of N number (defined by you) of messages and sends this collection of messages to the Clients on their initial connection.
* Sends notification to the Clients and safely closes all connections on application close.

**Task 1:**

Implement Client and Server using the following approach – one client – one thread. Read and write operations could be synchronous.

**Task 2:**

Implement Client and Server using one or both approaches:

* Classic async operations (BeginXXX/EndXXX) and threads pool for operations
* Task Parallel Library

**Score board:**

1-3 stars – 5 of 6 required tasks completed and implementation meets all requirements.  
4 stars – All 6 required tasks completed, all tests are GREEN and implementation meets all requirements.  
5 stars – No major remarks related to clean code principles (SOLID, KISS, DRY, etc.) / Optional task is covered.

*When you finish, please attach zip file or link to git and change the assignment status from "Planned" to "Need(s) review".*