Visual F# Tips:

- When writing concurrent programs, it is common to have some threads to run forever, or to stop and wait for something that never happens. Generally you can stop these by right-clicking in the interactive window and choosing Cancel Evaluation, or if this doesn't work, Reset Session. If neither works, you can use the Windows Task Manager to “End” the process fsi.

- When writing your code, it is normal to have some issues highlighted in with a red underline when a definition is not yet complete.

- Once it is complete, hover the mouse over any underlined parts to see feedback on the issue detected with that part of the code.

- Basic syntax and indentation issues are usually easy to fix, but typo errors can be more difficult – pay close attention to the two types reported when you hover. One is the actual type of an expression (or pattern, etc.) while the other is the type expected by the context. Use the types to guide you fixing the expression or the context so that the two types match.

- More generally, when writing code you should always be thinking in terms of the types of the various expressions, patterns, etc., and how they fit together. Hover over the variable names in an expression to see their types, and use these to guide the code you write.

- If you have syntax issues highlighted one part of your F# file (usually because you haven’t completed that part yet), but you’re currently on another unrelated part, detection and underlining of issues and hovering often won’t work which will make coding more difficult. Comment out the parts with syntax issues to fix this.

Lab details

This lab involves experimenting with concurrent code that requires threads to wait for each other in order to work correctly. This is usually achieved using monitors which are a form of locking or (also called mutual exclusion). The basic idea is that a thread can lock an object, and that only one thread can hold the lock on an object at any particular time – other threads requesting the lock on an object will wait until the thread holding the lock releases it.

In F# the standard way to request a lock is write a function, that should be run while the lock is held. This function is then passed to the library function

\[
\text{lock : 'a -> (unit -> 'b) -> 'b}
\]

\[
\text{lock objectToLock (fun () -> doThisWhileLocked...)}
\]

\[
\text{lock objectToLock <| fun () -> doThatWhileLocked...} \quad // \text{sometimes looks tidier}
\]

Often a thread holding a lock will encounter a situation where it needs to wait for the locked object to be modified by another thread. This is done via System.Threading.Monitor.Wait which releases the lock on an object (which must be held) and then waits until another thread (which must hold the lock) to call Monitor.PulseAll to indicate a modification involving the object. The original thread then waits to reacquire the lock, and continues.

The first exercise involves experimenting with a standard example of monitors called Producers and Consumers, to see why locking are waiting are required. The second involves adding appropriate locking and waiting to a simple implementation of bank accounts.
1. The starting point code for Q1 involves some threads producing numbers and one thread consuming them. The way the numbers are produced and consumed is not particularly interesting in this example - the most important feature is a general implementation of a buffer that stores produced items and supplies them to consumers. This is implemented as a circular buffer – using an array that wraps around to the start when the end is reached, with the position of the next item to be consumed moving forwards through the array (and similarly for the item most recently produced).

a) Test this program, and verify that the consumer always receives exactly the items that are produced, and in the same order that they were produced.

b) Try removing the synchronization code completely – all uses of lock, wait and wakeWaiters. Try running a few times. What goes wrong? Can you make this code work correctly without using locks? (It needs to work correctly for every possible timing.)

(Note, the calls to sleepABit are deliberately placed to make the effect of different possible timings more obvious, particularly on single core machines.)

c) Try also removing the loops that wait for count to be positive/less-than-n. What else goes wrong now?

2. The starting point code for Q2 creates two objects representing bank accounts with operations for withdrawals, deposits, and transfers to other accounts. Each account then transfers $100 to the other account 100 times (with a few sleeps to make interesting interleavings more likely).

a) Run this program a few times. Observe the final balances in the accounts – are they consistent with a sequence of transfers occurring? Does each account end up with $1000, as it should? Is the total amount of money still $2000? What is going wrong?

b) Fix this code by adding appropriate uses of locking, similar to those in the code in Q1 - the withdraw and deposit methods should lock the account. Verify that the code always leads to a final result that is consistent by testing it a few times and checking the final balances.

c) Now, ensure that each account balance never becomes negative by adding appropriate uses of waitFor and wakeWaiters in the withdraw and deposit methods similar to in Q1. (Here, and in the rest of the question, you should ensure that the account class is suitable for use in other programs as well that may have more accounts, and do more than just transfers.)

d) Add a separate thread that prints the balances of both accounts periodically while the other threads are running. Have it print 10 times (I found that printing every 200ms worked well). Run the program a few times, and verify that the balance never goes negative.

e) Notice that the total amount of money printed is not always $2000 while the threads are running, and so is inconsistent – it looks like some money is missing. Fix this by locking both accounts for the whole of the transfer method. Also lock both accounts each time before printing in the printing thread. Include a call to sleepMaybe after acquiring the first lock – this will make some interesting interleavings more likely, so that you don't have to run the program a too many times.

f) Test the previous part a few times – and don't be surprised if the threads appear to stop and never complete (interrupt them as explained in the Visual F# Tips). This happens when two threads are each waiting for a lock the other has – this is called a deadlock. Try to figure out exactly what is going wrong by tracing through the steps, and/or adding print statements.

g) [Challenge] Fix the issues you identified in the previous part. This is not easy, but attempting this will help you understand some issues that will be important in the project. It may help to remove the waitFor and wakeWaiters calls to start with. (Hint: if threads which acquire two locks always acquire them in the same order, they will not deadlock.)

Ensuring that the balances don't become negative is particularly challenging - don't spend too much time if you can't see how to do it, but spend long enough to understand why it is difficult. (Hint: waitFor only releases the lock it is called on.)