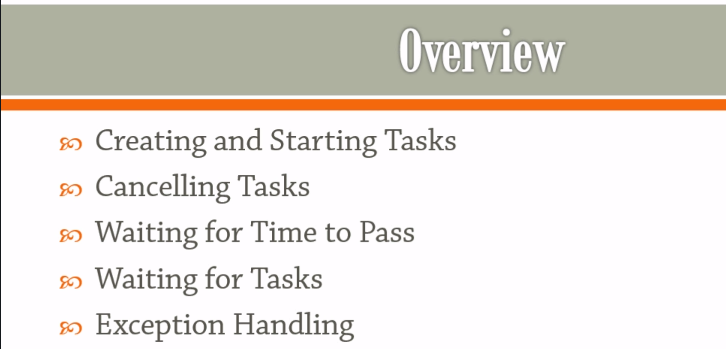
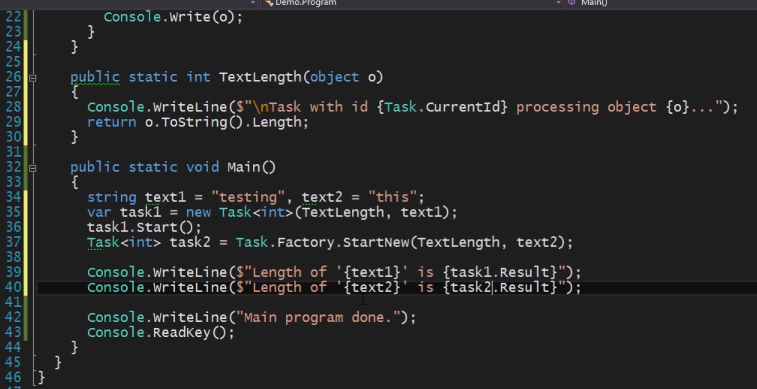
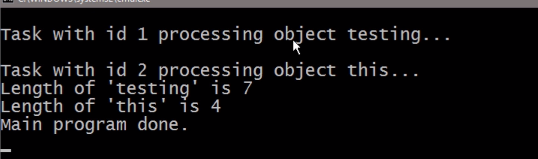
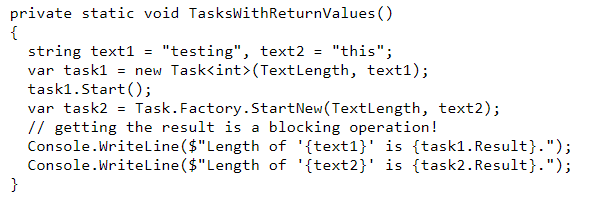
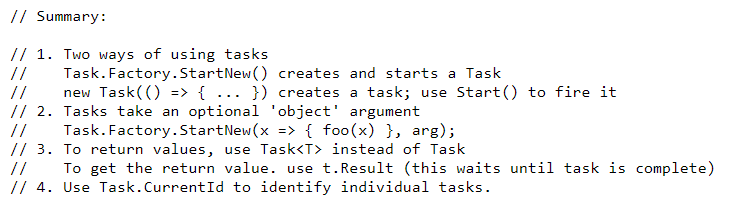
Learn Parallel Programming with C# and .NET

**Section 1: Task Programming**

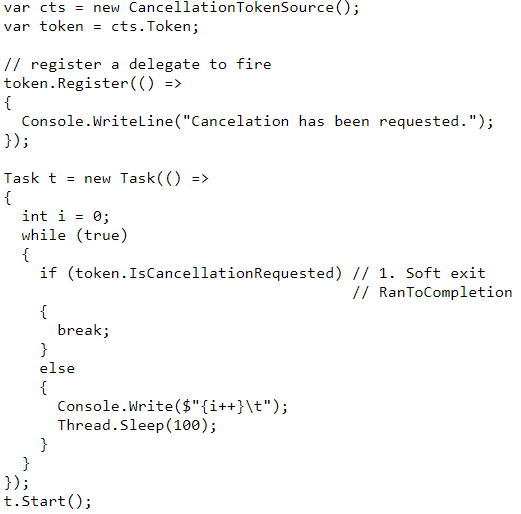
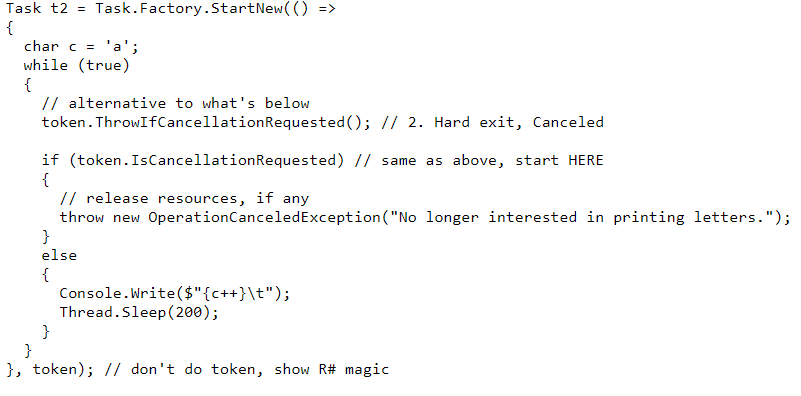
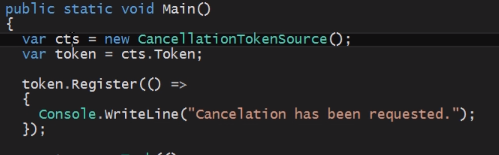
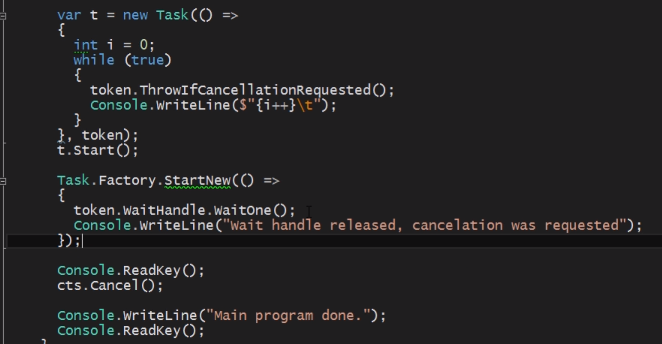
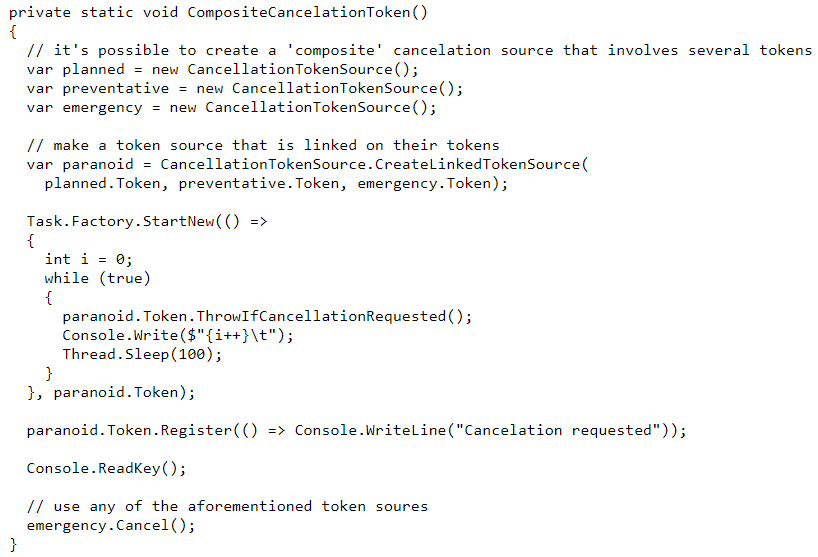
2. Overview:

* a task is a unit of work in the TPL
* 

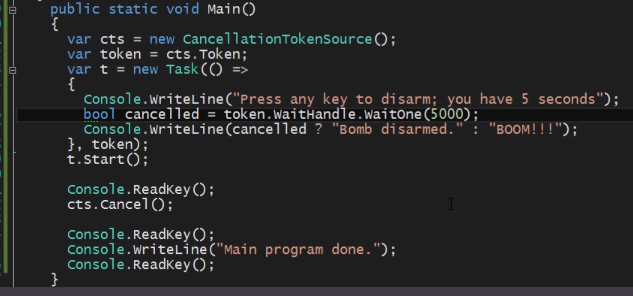
3. Creating and Starting Tasks:

* Question: So, what is a task and why do you want to use it?
* Answer: A task is just an abstraction. It just .Net ways of grouping a unit of work together and telling the scheduler, this unit of work can be executed on a separate thread.
* Example:
* 
* 
* As soon as you ask a task for its result, you're basically saying that you're prepared to wait until the task is finished which means that getting the result of a task is a blocking operation.
* That will basically block until the task is complete and when it's complete then it's going to give you the results.
* 
* 

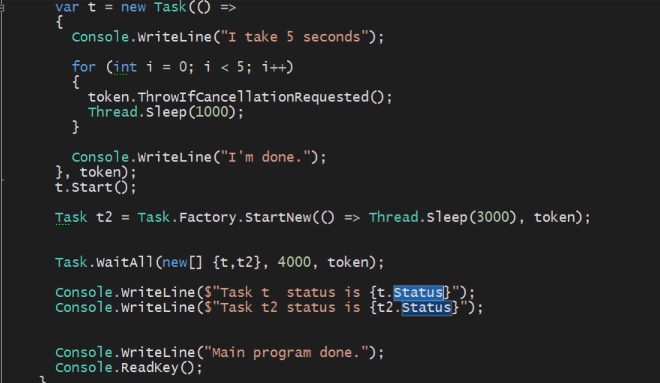
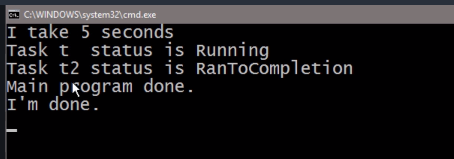
4. Cancelling Tasks:

* Tasks get executed on separate threads which means at some point you might want to cancel a particular task and TPL provides explicit mechanism for doing exactly that.
* In order to support cancellation, the task for library gives you two things. It gives you a cancellation token source and cancellation tokens.
* So, there are two ways of cancelling your execution. One of these is kind of soft failure. That is like using ‘break’ and ‘return’ statement with some conditions.
* 
* Other one is a canonical way: Instead of breaking throw an exception. This one is recommended.
* 
* Question: However, if you don't throw the exception the task isn't going to know that it's being canceled it's just going to tell you that it completed successfully. So, the question is whether or not you want to actually be notified of the fact that someone cancel the task. It's really a design decision?
* However, this is the canonical way this is the way that TPL recommends you do because then you have a record of the fact that somebody canceled this particular task and also when you have several tasks tied to a single token you can better figure out what happened whether the task completed successfully or whether somebody generally canceled.
* Question: One question that we might want to answer is how we can monitor when a task is canceled. How can we be notified about it?
* Answer: once again there are different ways of doing exactly that. , one of those ways is you can basically subscribe to an event that's available right on the tokens:
* 
* Other way which is also very interesting is to basically wait on the cancellation tokens wait:
* When you call register, you're subscribing to an event. Here the idea is that you take the tokens waitHandle explicitly and you say I'm going to wait. I'm going to wait on this handle until somebody cancels the token until somebody actually invokes the cancellation. So as soon as cancellation is invoked this function call which is actually blocking will release and will allow you to do whatever it is you want to do.
* 
* So that's two of the ways in which you can actually monitor that somebody wants to essentially cancel the task.
* Composite cancellation tokens:
* So essentially the idea is that you can have several cancellation tokens sources for example you might have a token source for cancellations which were planned.
* So, we have different cancellation token sources what we can do is we can now specify a token source which reacts to any of those three token sources being triggered essentially a kind of paranoid mode:
* So paranoid is going to be a cancellation token source which essentially if you get a token from paranoid that token will cause cancellation when you trigger when you call cancel on either planned or preventative or emergency.
* 
* because we have a token from a linked token source that will stop our execution.

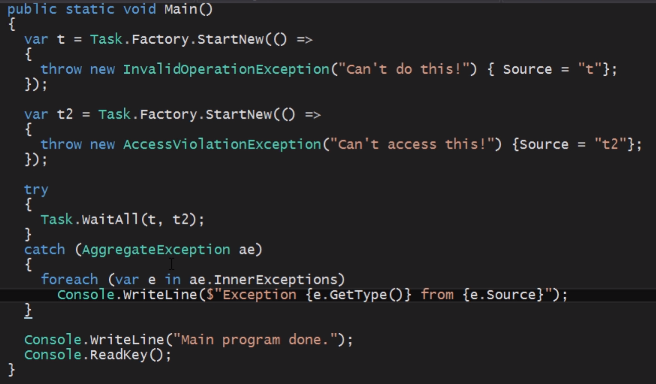
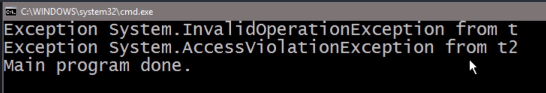
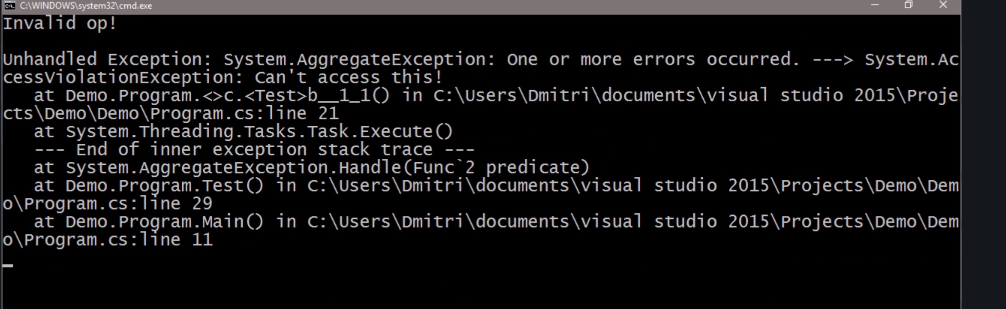
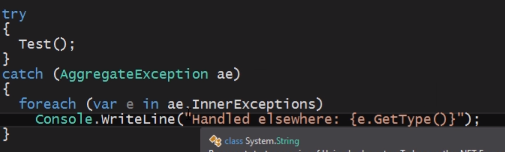
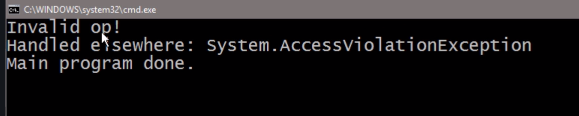
5. Waiting for Time to Pass:

* Thread.Sleep:
* So, if you do Thread.Sleep that basically just pauses the thread of execution.
* But it's not just that. Not only does it pause a thread of execution, it actually tells the scheduler that we don't need the processing time anymore so the scheduler can get another task executing as you sleep which is great.
* It's great for the purposes of efficiency because then you can get some other work done while this chunk of work obviously doesn't want to get itself done.
* Thread.SpinWait and SpinWeight.SpinUntill:
* Spinwait means that you also pause the thread but there is a different rule here.
* Essentially you don't give up your place in the execution task. So essentially what's happening is the scheduler isn't going to get some other task executing while you are spinning. Instead you are basically wasting CPU cycles but you're not giving up your turn in the overall execution scheme which means that on the one hand you are wasting resources obviously on the other hand you're avoiding context switching.
* So, if you need to wait just a little bit then Spinwait might be more efficient than Thread.Sleep.
* WaitHandle on CancellationToken:
* 
* So essentially what this means is that you can take the tokens weight handle and you can basically wait for cancellation a certain amount of time and then check whether in this amount of time did someone actually cancel or not and if they did cancel then you get a true. If they didn't count so you get a false and then you can respond to that.

6. Waiting for Tasks:

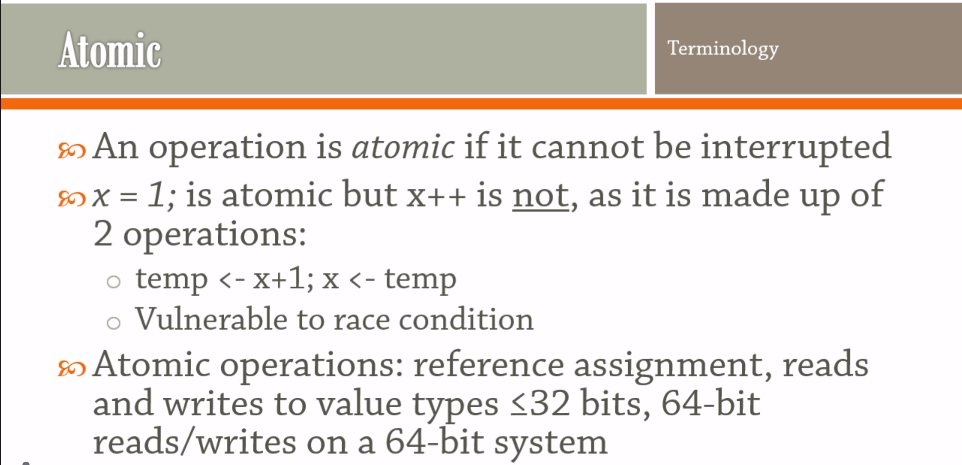
* Task.WaitAny: It waits for any of the tasks to complete as soon as one of those tasks is done.
* 
* 

7. Exception Handling:

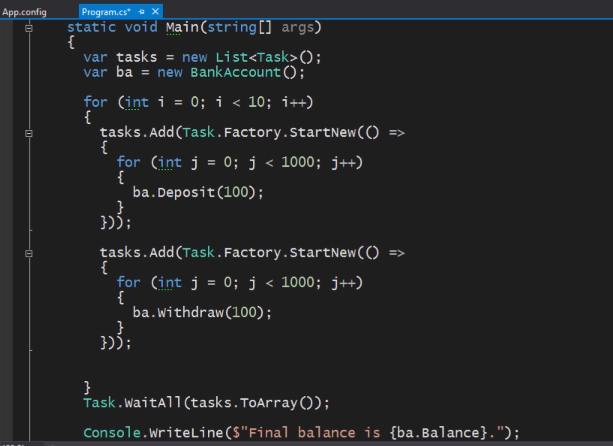
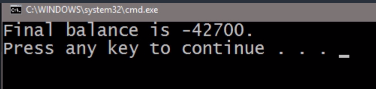
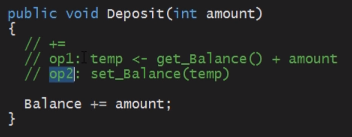
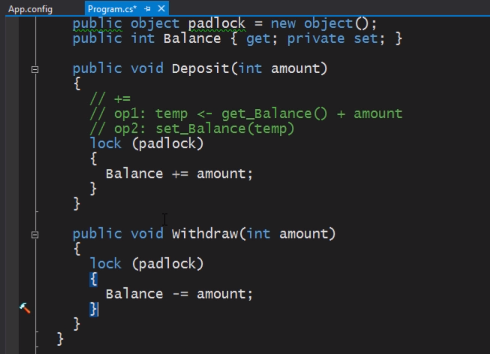
* different tasks can throw different types of exceptions. if we don't handle an exception in a separate task there is going to be ignored completely.
* AggregateException:
*  
* We have an even more powerful mechanism for handling on the part of the aggregate exception and propagating the rest of the aggregate exception up to the hierarchy.
* Now what I can do is that I'm only prepared to handle the invalid operation exception:
* 
* So, if I now execute this the program will crash once again:
* 
* We can see that the program has crashed with the aggregate exception, even though it seems as we caught the entire set of exceptions.
* What we did is we explicitly specify that we're only prepared to handle the invalid operation exception and the other exceptions propagated up to the stack. And that we can handle like below:
* 
* 
* The other exception is handled elsewhere(caller function handled other exceptions).

**Section 2: Data Sharing and Synchronization:**

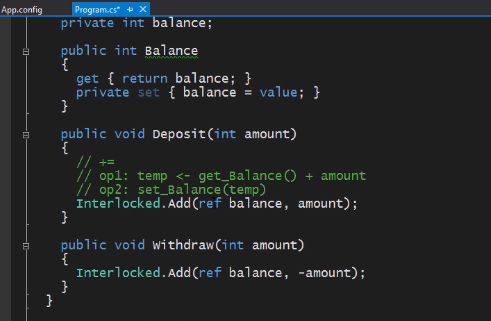
9. Overview:

* This is all about how you control access to data. When you've got several threads reading the data or indeed writing the data.
* Critical sections are the areas of code whose access is controlled by a lock object.
* Then we'll talk about interlocked operations and how those can be used with certain primitive types to avoid using locks.
* 
* **Atomicity:** So essentially, we call an operation atomic if this operation cannot be interrupted meaning it cannot be separated into several different parts such that between the execution of one part and another, some other thread comes in and does something. So, an operation is atomic. If it cannot fundamentally be interrupted by something else.
* 
* For example, reference assignment when you have a variable of a reference type and you assign something to it that's an atomic operation that cannot be interrupted by anything.
* generally, reads and writes to value types such as an int or short or a Boolean OR when the value type is less than or equal to 32 bits, they are in fact the atomic.
* And if you're on the 64-bit system then 64 bit reads, and writes are also atomic.
* But if you're on a 32-bit system then there is no atomic Guarantee for 64 bit types and certainly not for larger types than 64 bits as well.

10. Critical Sections:

* 
* For above program final balance should be zero. But see the output:
* 
* So, each time we get a different output sometimes may be 0 also.
* Question: So, let's ask ourselves why this is happening?
* Answer: And the reason why this is happening is because the withdraw and deposit operations, they are not atomic.
* Question: Why this operation is not atomic?
* Answer:
* So, these are two operations, we’re essentially getting something happening in between those operations and messing up the overall result.
* 
* Solution: So, the simplest setup for getting this to actually work correctly is to set up a critical section.
* Now a critical section is basically a piece of code or a marker around the piece of code which says that only one thread can enter this particular area.
* 

11. Interlocked Operations:

* interlocked class contains atomic operations on variables.
* So now that we're using Interlocked class, it's similar to what we have achieved with the locking but now we're doing this rather more direct mechanism that is so-called lock free programming.
* 
* Interlocked.MemoryBarrier(): It is a shorthand for a Thread.MemoryBarrier.
* Now one of the problems with concurrent code is that sometimes the CPU can actually reorder instructions which are written by us.
* So, what the memory barrier does is it basically tells the CPU that in no instance can the instructions appear before the memory barrier be executed in the block after the memory barrier. So that's the goal of the memory Barry here.

12. Spin Locking and Lock Recursion:

* The idea that we continue waiting on something, but you don't give up your position in the scheduler, so you don't yield to other threads to only be rescheduled later on. Instead you're just wasting C-p cycles.
* So, there is a construct called a SpinLock which does something similar to the kind of locking that we've seen previously but it does it using this spinning mechanism and it also introduces us to another interesting idea or interesting problem called LockRecursion.
* Pending Study Letter…….