Intro to Async and Parallel Programming in .NET 4:

Working with Tasks: Creating , Waiting, and Harvesting Results

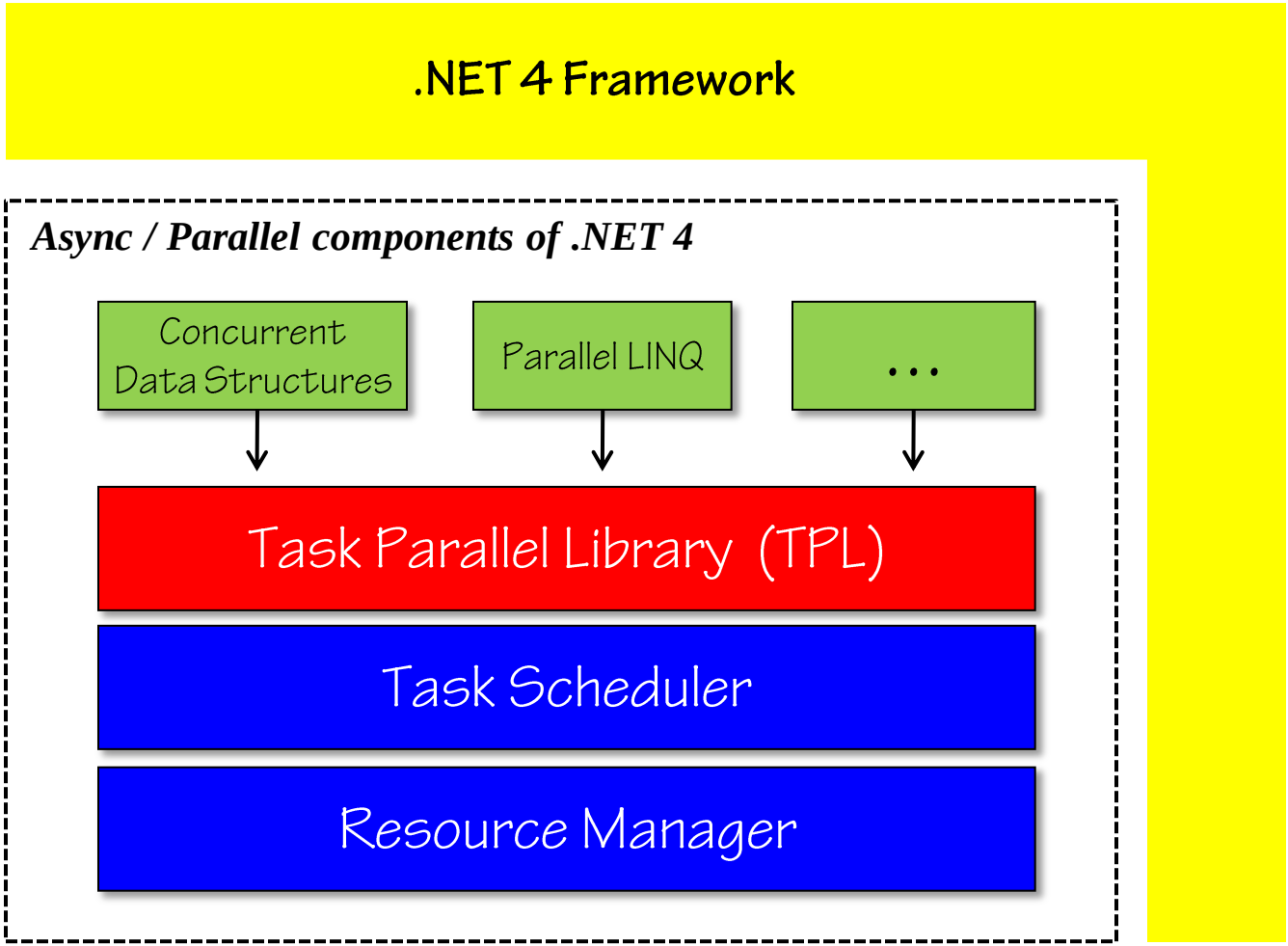
The most common operations on tasks…

Overview

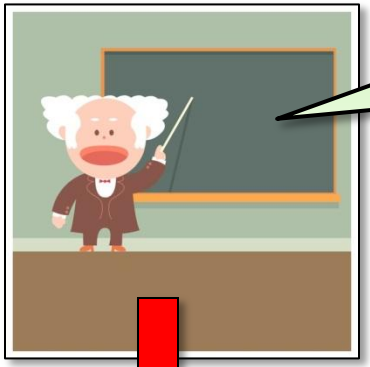
- **Your presenter: Joe Hummel, PhD**
  - PhD in field of high-performance computing
  - drjoe@pluralsight.com

- **Agenda for this module:**
  - Tasks redux (creating, code vs. façade, etc.)
  - How to…
    - *Wait for a task to finish*
    - *Wait for one or more tasks to finish*
    - *Harvest a task result*
    - *Compose tasks*

Technologies in .NET 4



Review — what's a task?



Task == *object representing an ongoing computation.*

Task object *provides a means to check status, wait, harvest results, store exceptions, etc.*

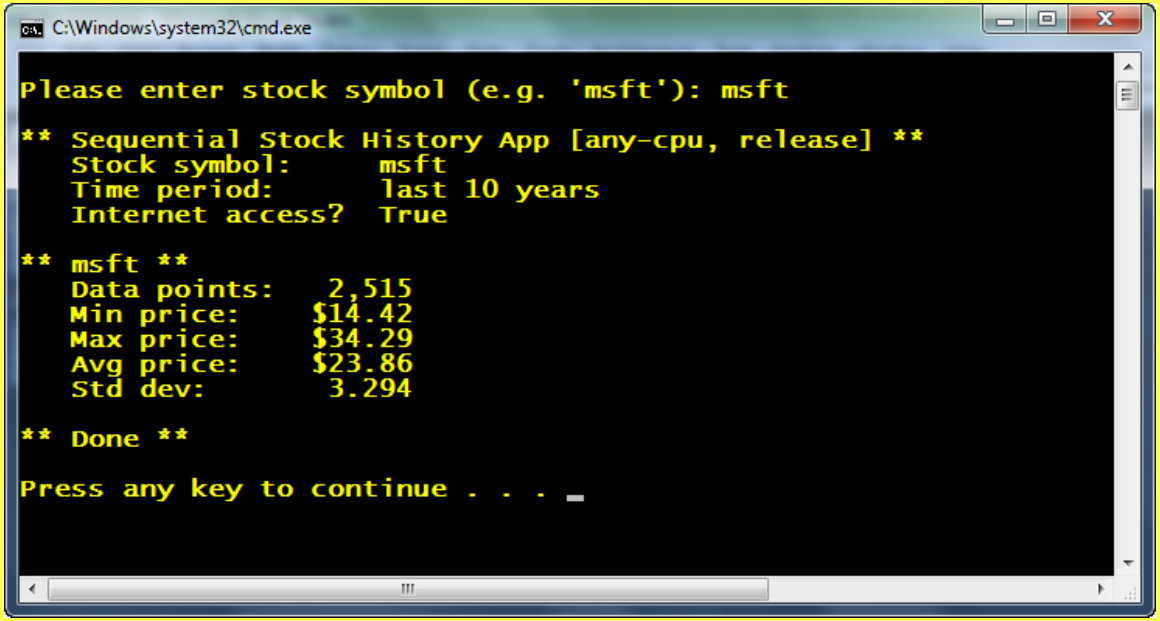
```
// "code" task to execute given op:
Task T1 = Task.Factory.StartNew(() => { /*code*/ });

// "façade" task over existing op:
var  op = new TaskCompletionSource<T>();
Task T2 = op.Task;
```

Task
Status: ___
Result: ___
Exception: ___
…

Task
Status: ___
Result: ___
Exception: ___
…

DEMO

- Code and Façade tasks
- Downloading and processing historical stock data

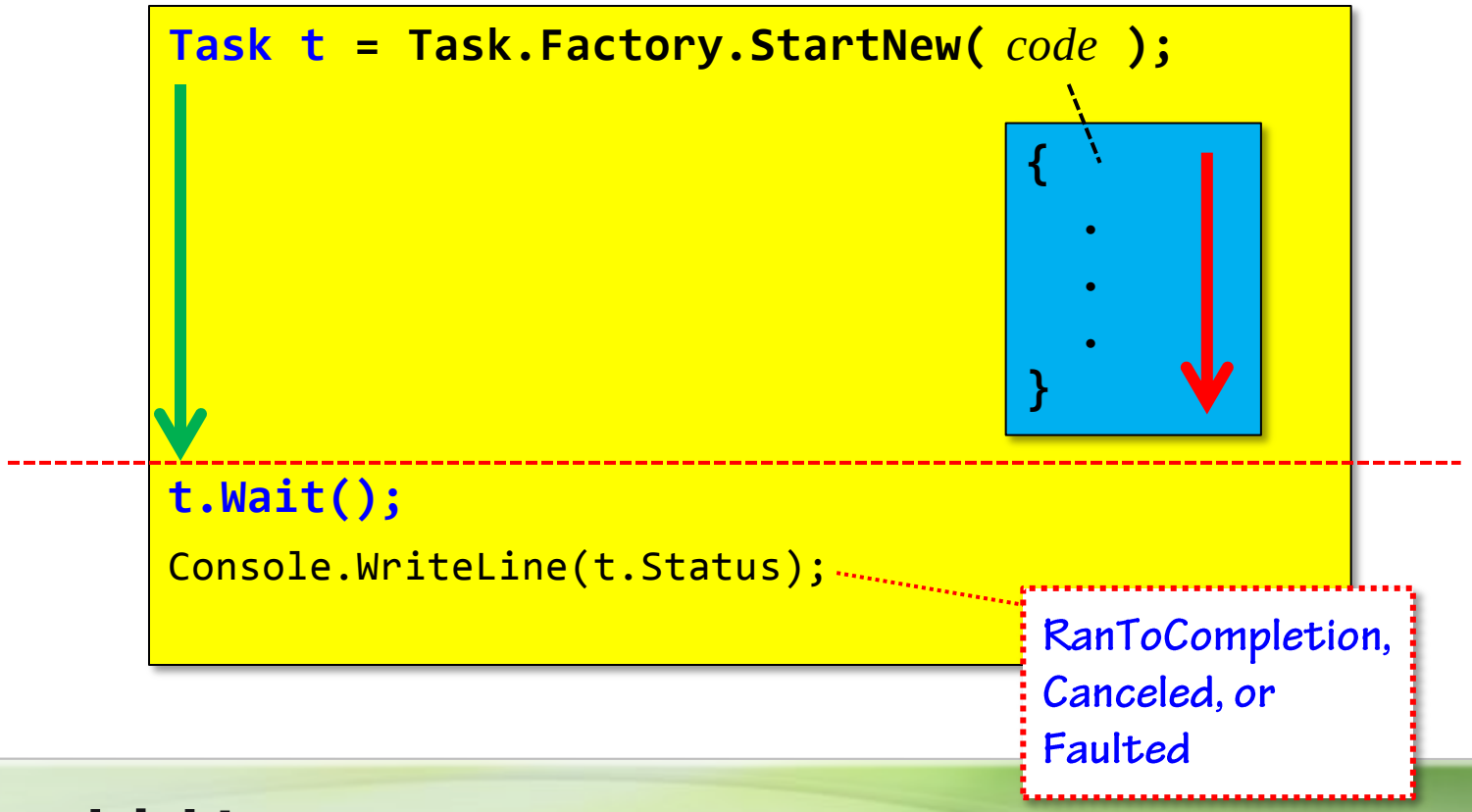


How to

- Wait for a task to finish
- Wait for one or more tasks to finish
- Return a value from a task
- Compose tasks

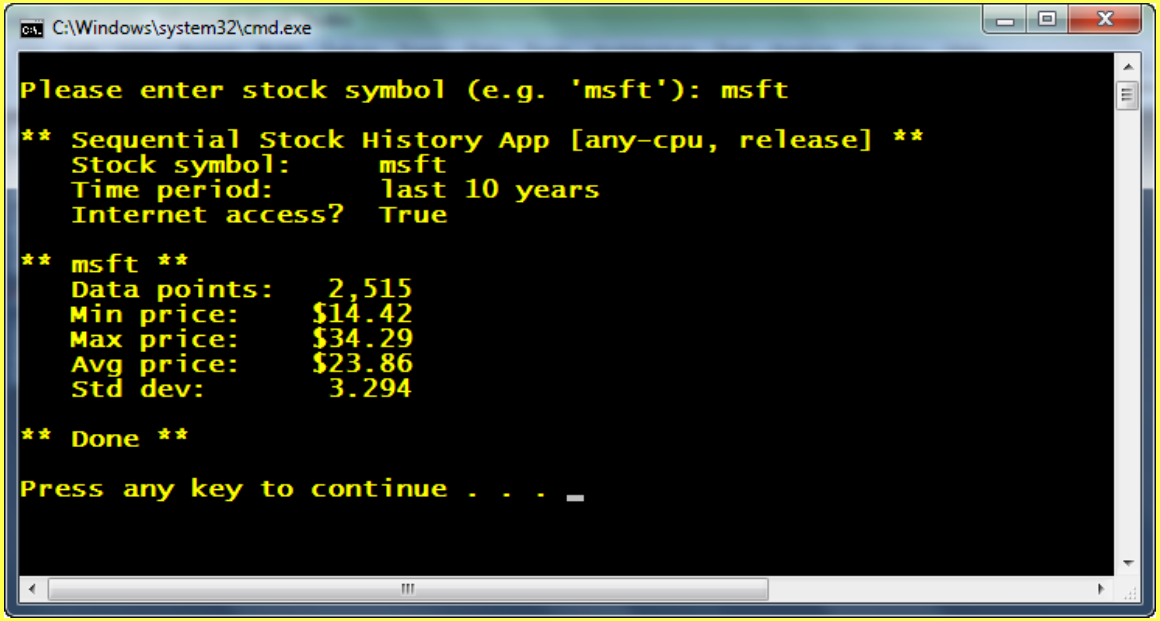
Waiting

- Need to wait for a task to finish?
- Call **.Wait** on task object…



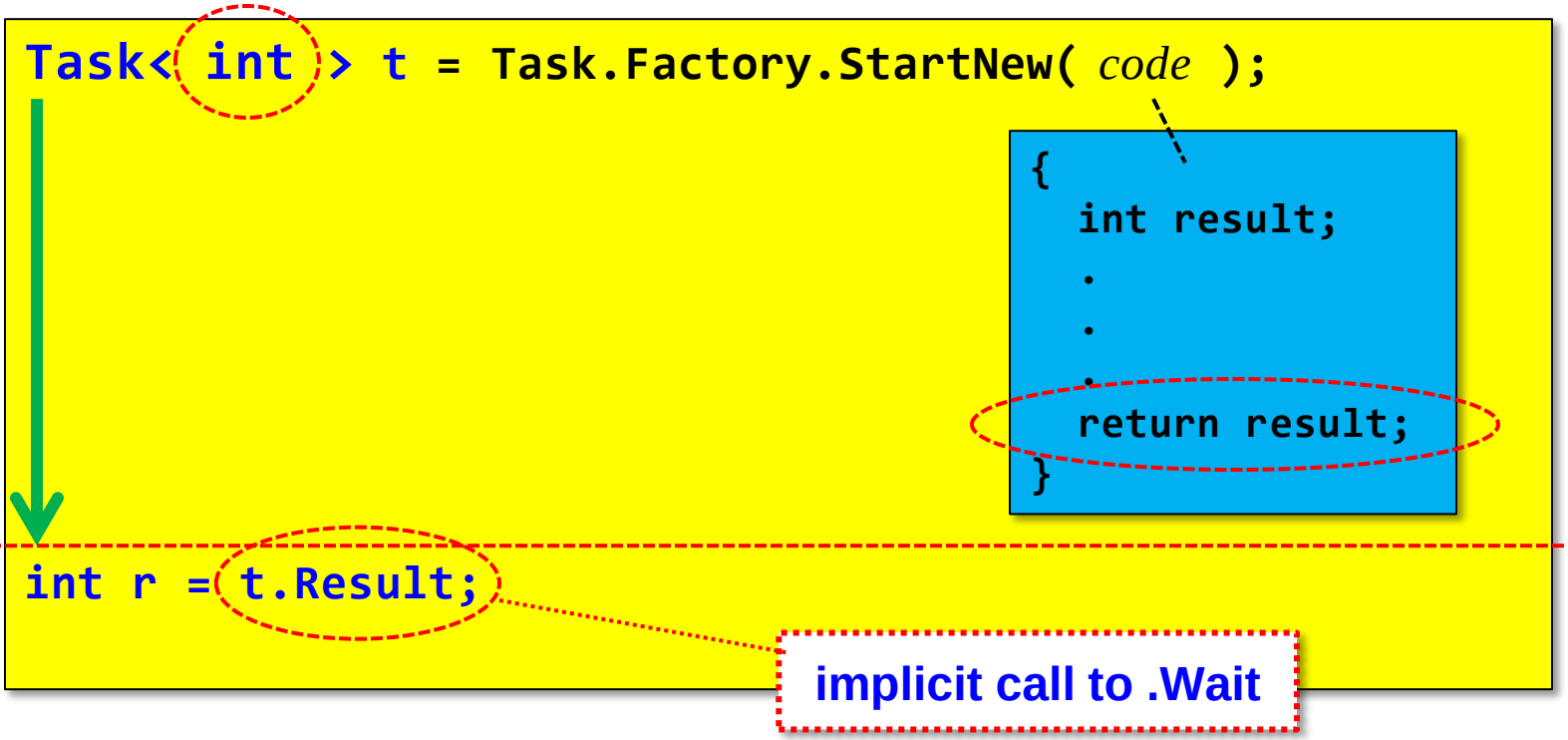
DEMO

- Waiting for tasks to finish
- Historical stock data revisited…



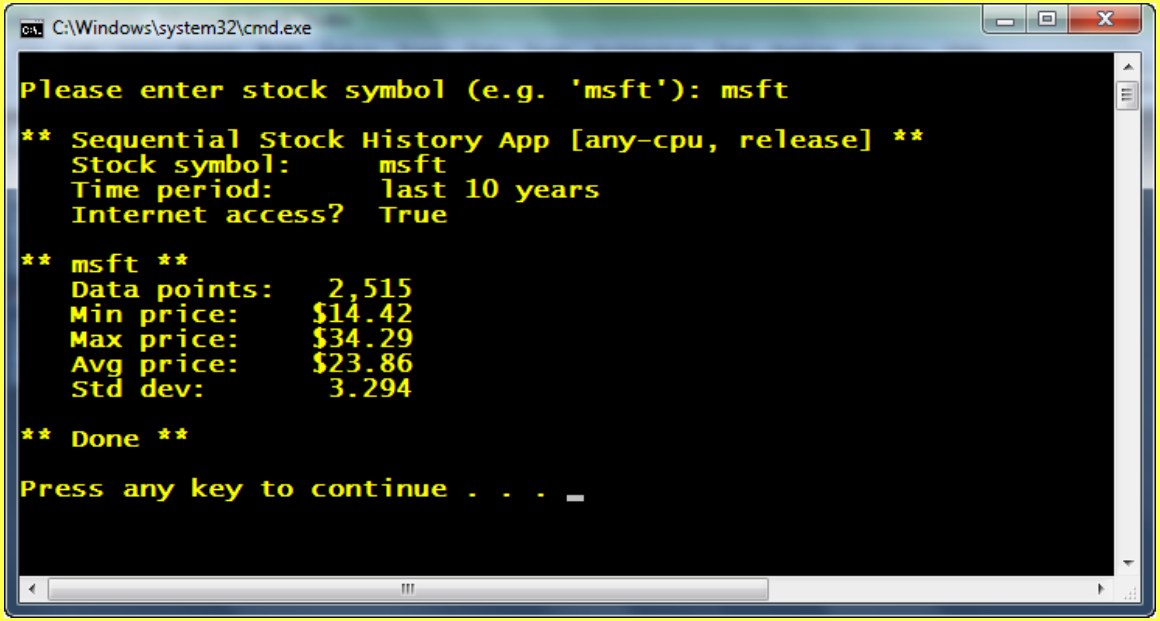
Harvesting results

- Is task computing a result?
- Specify **type** when creating task
- Harvest value via **.Result**



DEMO

- Harvesting task results
- Historical stock data revisited…



Waiting on multiple tasks

- **What's the best way to wait for a *set* of tasks to finish?**
  - *Calling .Wait on each task implies an ordering that might not hold*
- **What's the best way to wait for the *first* task to finish?**
  - *e.g. parallel search of multiple sites, first one wins*

```
Task t1 = Task.Factory.StartNew( code );
Task t2 = Task.Factory.StartNew( code );
Task t3 = Task.Factory.StartNew( code );

Task[] tasks = { t1, t2, t3 };
 .
 .
 .
```

```
Task.WaitAll( tasks );   // wait for ALL to finish:
```

```
int  index = Task.WaitAny( tasks );   // wait for FIRST to finish:
Task first = tasks[index];
```

"*WaitAllOneByOne*" pattern

- Wait for **all** tasks to finish, but process results as each **one** completes:

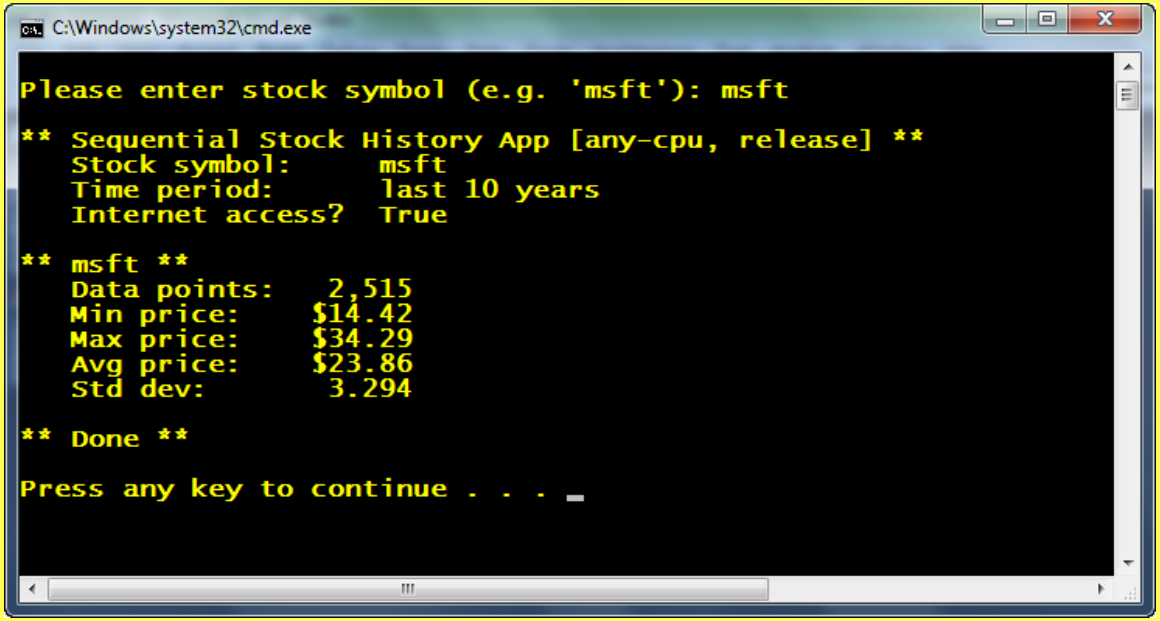
```
List< Task<TResult> >  tasks = new List< Task<TResult> >();

for (int i=0; i<N; i++)  // Start N tasks:
  tasks.Add( Task.Factory.StartNew<TResult>(code) );


while (tasks.Count > 0)  // Wait for all, one-by-one:
{
  int index = Task.WaitAny( tasks.ToArray() );
    .
    .  // process tasks[index].Result:
    .
  tasks.RemoveAt(index);
}
```

Use this pattern when:

1. *Some may fail — discard / retry*
2. *Overlap computation with result processing — aka hide latency*

DEMO

- WaitAll and WaitAny
- Historical stock data revisited…



Task composition

- The completion of one task triggers the start of another:

```
Task T1 = Task.Factory.StartNew( () =>
  {
    .
    .
    .
  }
);

Task T2 = T1.ContinueWith( (antecedent) =>
Task T2 = Task.Factory.StartNew( () =>
  {
  T1.Wait();
    .
    .
    .
  }
);
```

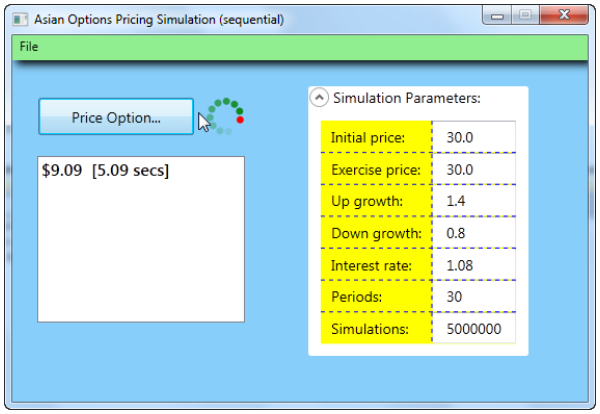
parameter denotes task that just completed — i.e. T1 in this case

Implicit .Wait — T2 does not start until T1 completes

Why? Allows .NET to optimize scheduling…

Example

- **From previous lecture:**
  - Task T1 runs simulation, then T2 updates GUI with result via UI thread




```
Task<string> T1 = Task.Factory.StartNew( () =>
  {
    .
    .   // code to run simulation:
    .
    return result;
  }
);
```

```
Task T2 = T1.ContinueWith( (antecedent) =>
  {
    string result = antecedent.Result;
    .
    .   // code to update UI:
    .
  },
  TaskScheduler.FromCurrentSynchronizationContext()
);
```

Variations

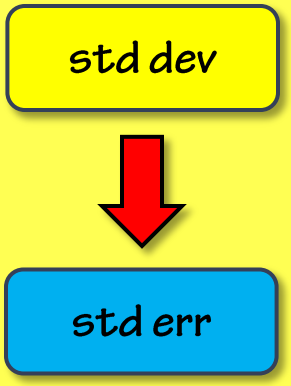
- TPL also supports many-to-one compositions…

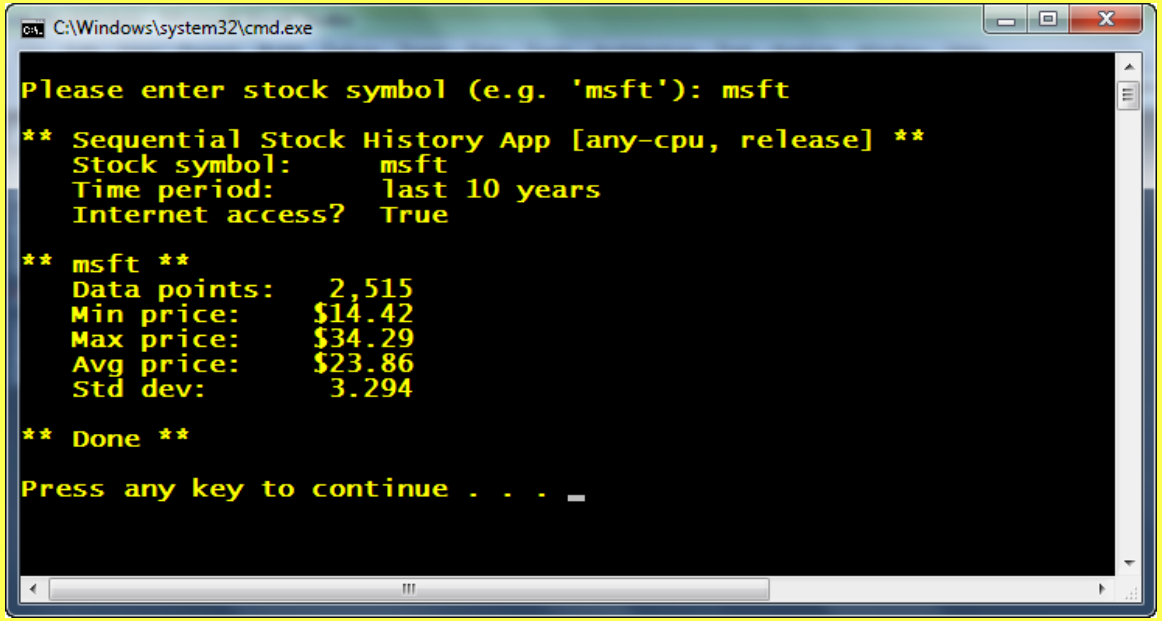
```
Task t1 = Task.Factory.StartNew( code );
Task t2 = Task.Factory.StartNew( code );
Task t3 = Task.Factory.StartNew( code );

Task[] tasks = { t1, t2, t3 };
```

```
Task.Factory.ContinueWhenAll(tasks, (setOfTasks) =>
{
   .
   .   // continuation code when all have finished:
   .
});

Task.Factory.ContinueWhenAny(tasks, (firstTask) =>
{
   .
   .   // continuation code when first has finished:
   .
});
```

DEMO

- Computing stderr using .ContinueWith
- Historical stock data revisited…





Summary

- **Task Parallel Library provides lots of support for working with tasks:**
  - *Waiting*
  - *Harvesting results*
  - *Composition*
  - *Exception handling*
  - *Cancellation*

  } next lecture!

- **This support works uniformly for both:**
  - **code** tasks
  - **façade** tasks

References

- **Microsoft's main site for all things parallel:**
  - http://msdn.microsoft.com/concurrency
- **MSDN technical documentation:**
  - http://tinyurl.com/pp-on-msdn
- **I highly recommend the following short, easy-to-read book:**
  - *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*, by C. Campbell, R. Johnson, A. Miller and S. Toub, Microsoft Press

    Online: http://tinyurl.com/tpl-book