Intro to Async and Parallel Programming in .NET 4:

Tasks and Task-based Programming for Responsiveness and Performance

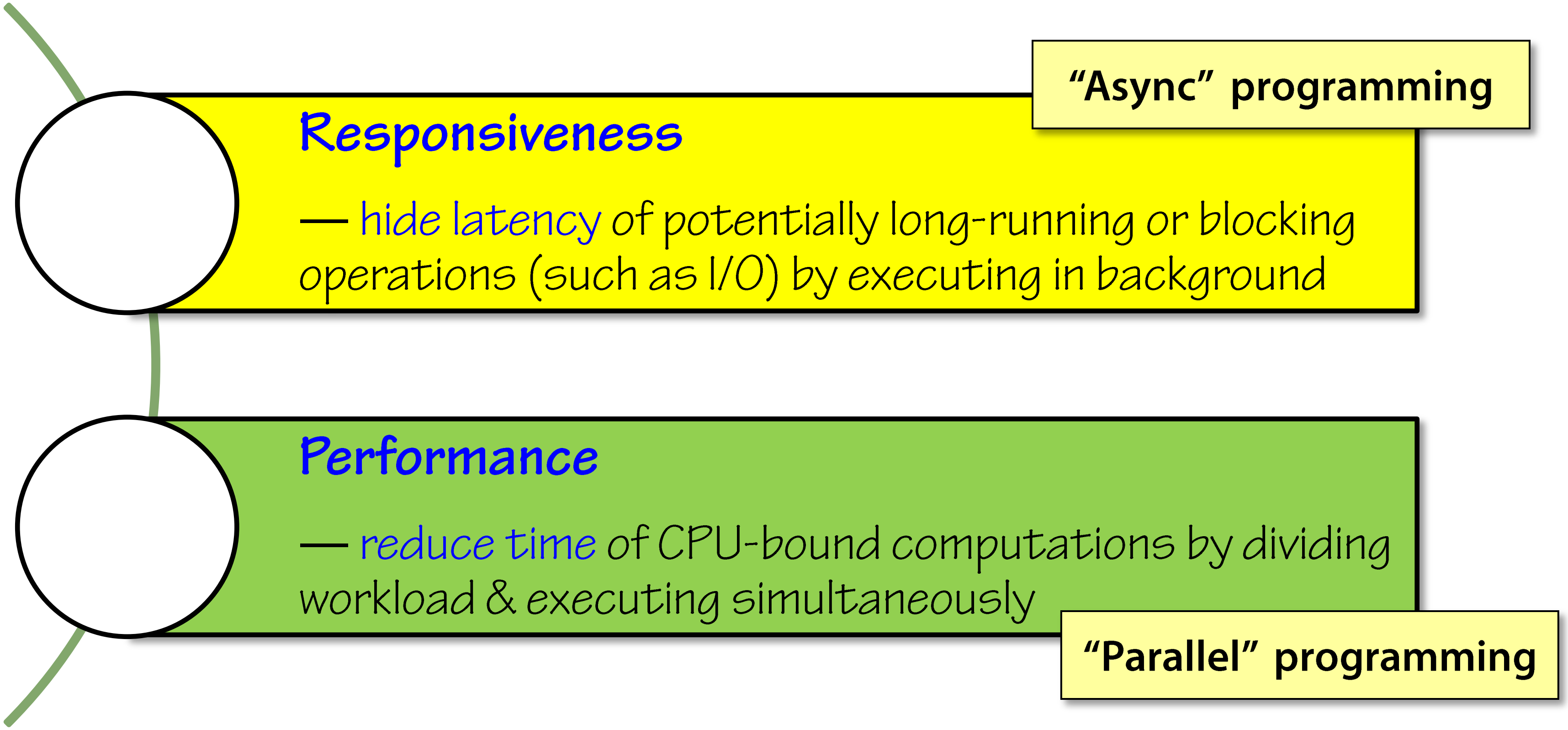
A quick intro to using tasks for async and parallel programming

pluralsight
see what you can learn

Overview

- **Your presenter: Joe Hummel, PhD**
  - PhD in field of high-performance computing
  - drjoe@pluralsight.com
- **Agenda for this module:**
  - Why are we here?
  - What's a task?
  - Creating tasks
  - Asynchronous programming with tasks
  - Parallel programming with tasks

Motivation



"Async" programming

Responsiveness

— hide latency of potentially long-running or blocking operations (such as I/O) by executing in background

Performance

— reduce time of CPU-bound computations by dividing workload & executing simultaneously

"Parallel" programming

Async and Parallel Programming

- **Based on Tasks and Task Parallel Library (TPL)**
  - Available now in .NET 4
  - Coming soon to Silverlight 5
- **Why another approach? We already have:**
  - *Threads*
  - *Async Programming Model* (e.g. async delegate invocation)
  - *Event-based Async Pattern* (e.g. BackgroundWorker class)
  - *QueueUserWorkItem*

New model is evolutionary:

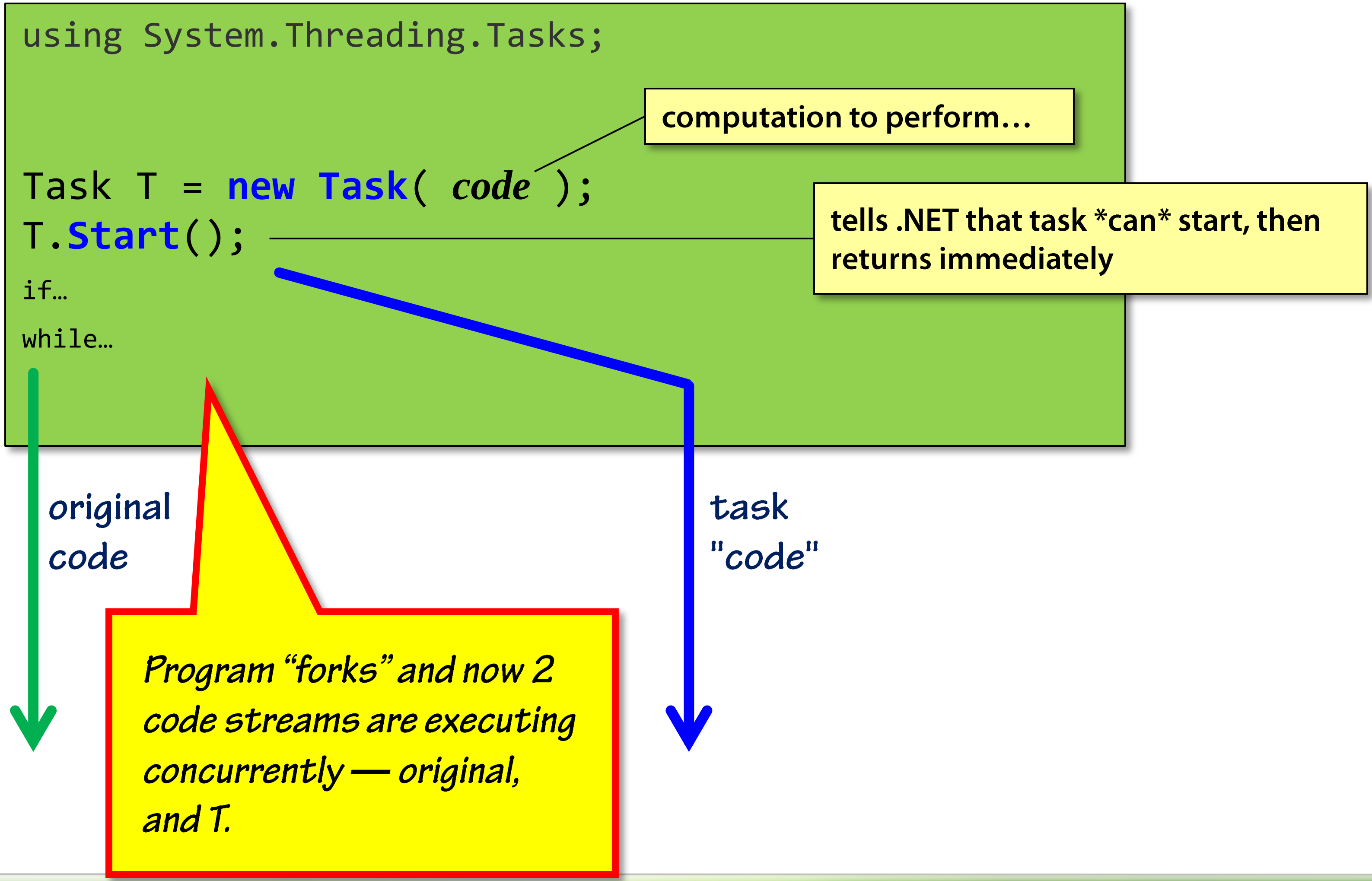
1. *Canceling*
2. *Easier exception handling*
3. *Higher-level constructs*
4. *And more…*

Tasks

- Programming model is based on the concept of a **Task**

Task == *a unit of work; an object denoting an ongoing operation or computation.*

Creating a task



```
using System.Threading.Tasks;


Task T = new Task( code );
T.Start();
if…
while…
```

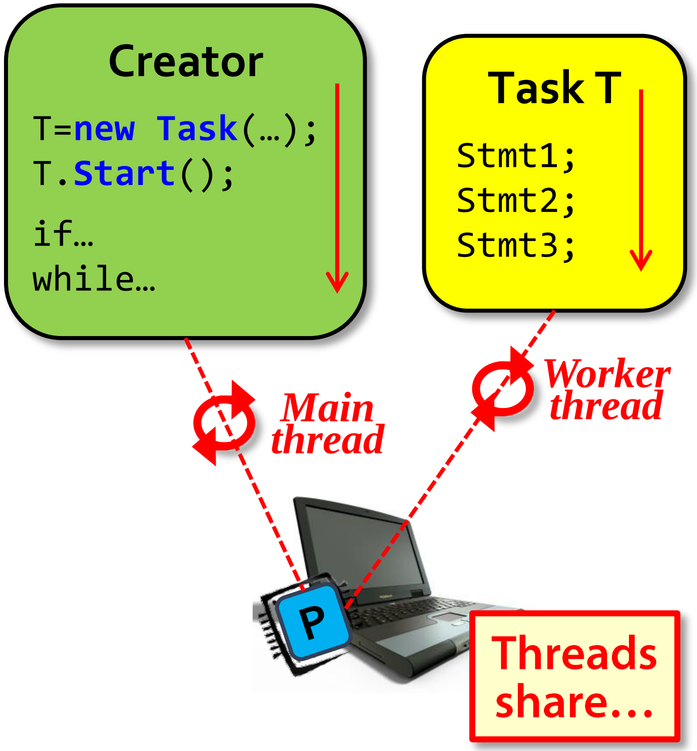
computation to perform…

tells .NET that task *can* start, then returns immediately

Program "forks" and now 2 code streams are executing concurrently — original, and T.

Execution model — a quick look

- Code-based tasks are executed by a thread on some processor
- Thread is dedicated to task until task completes



Creator

```
T=new Task(…);
T.Start();
if…
while…
```

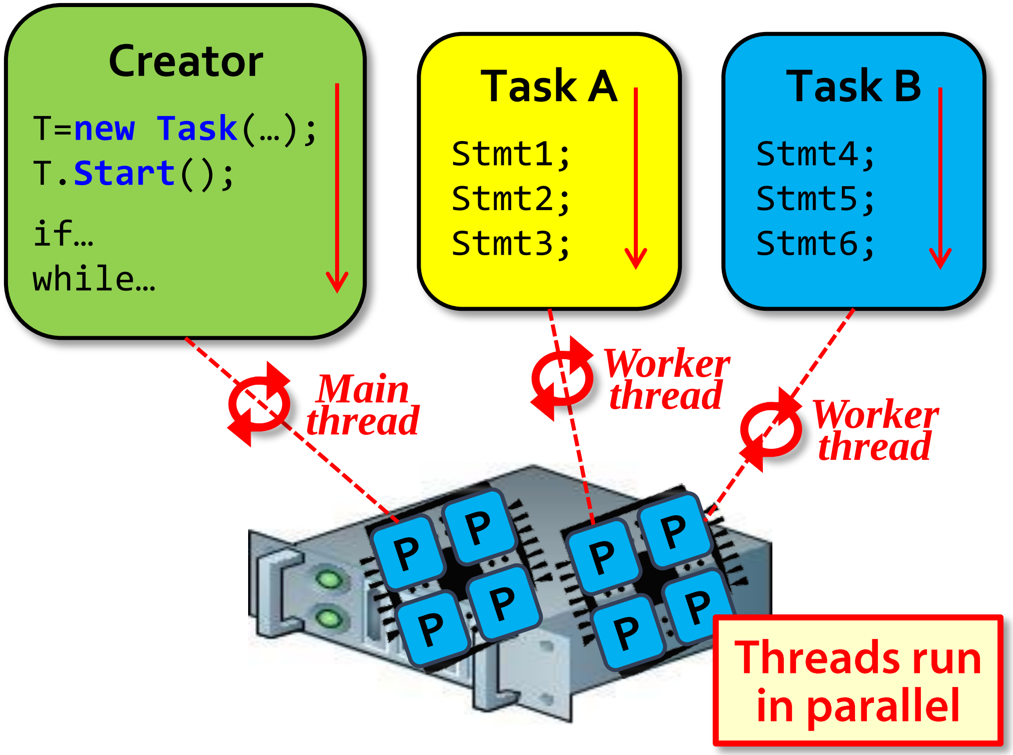
Task T

```
Stmt1;
Stmt2;
Stmt3;
```

Main thread

Worker thread

Threads share…



Creator

```
T=new Task(…);
T.Start();
if…
while…
```

Task A

```
Stmt1;
Stmt2;
Stmt3;
```

Task B

```
Stmt4;
Stmt5;
Stmt6;
```

Main thread

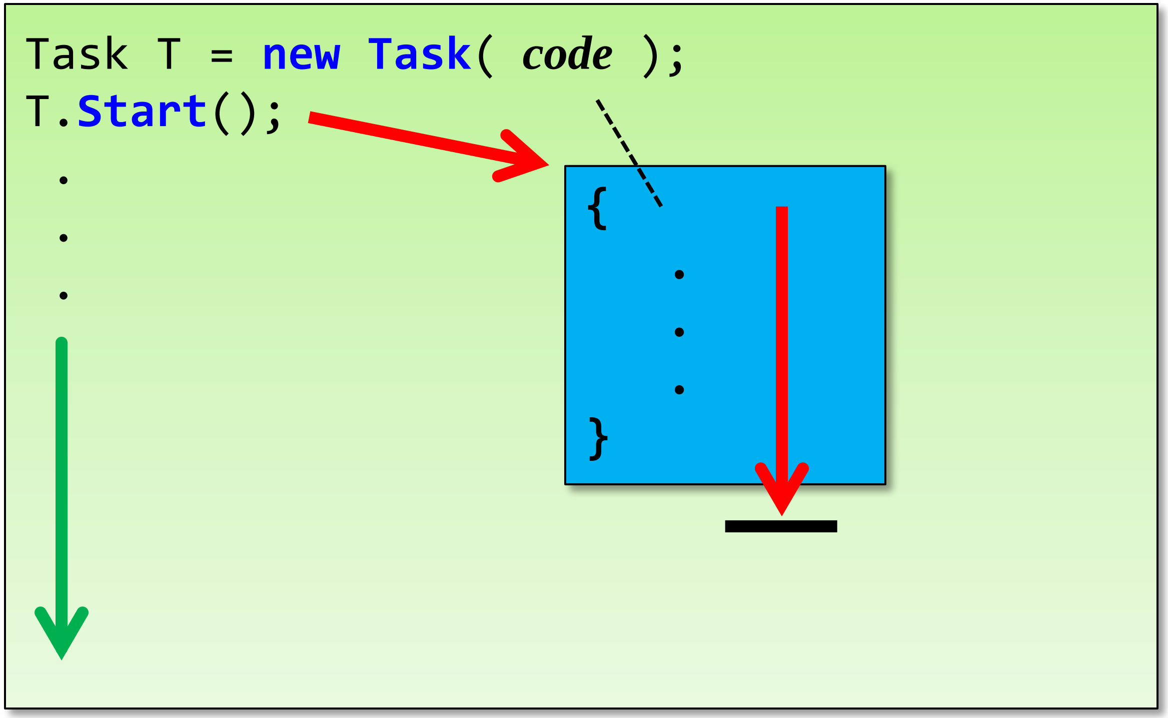
Worker thread

Worker thread

Threads run in parallel

Task completion

- **When does a code task complete?**
  - When code block is exited — naturally, or by throwing an exception

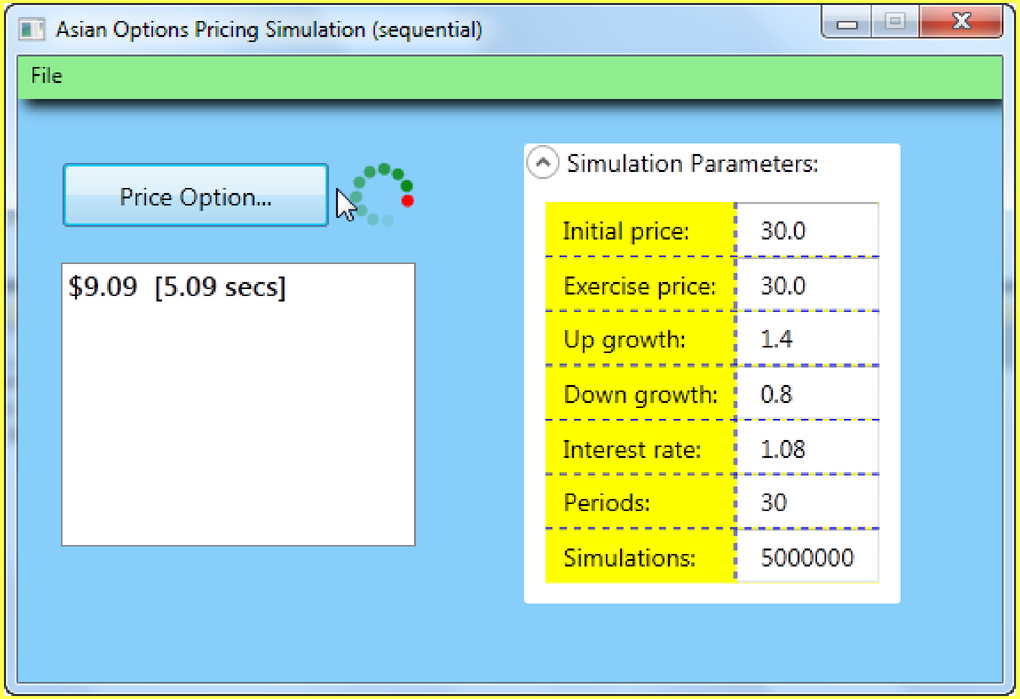


```
Task T = new Task( code );
T.Start();
  .
  .
  .
```

```
{
    .
    .
    .
}
```

DEMO

- Asynchronous programming for responsiveness
- Asian Options finance modeling application



Asian Options Pricing Simulation (sequential)

$9.09 [5.09 secs]

Simulation Parameters:

Initial price:	30.0
Exercise price:	30.0
Up growth:	1.4
Down growth:	0.8
Interest rate:	1.08
Periods:	30
Simulations:	5000000

Demo Summary

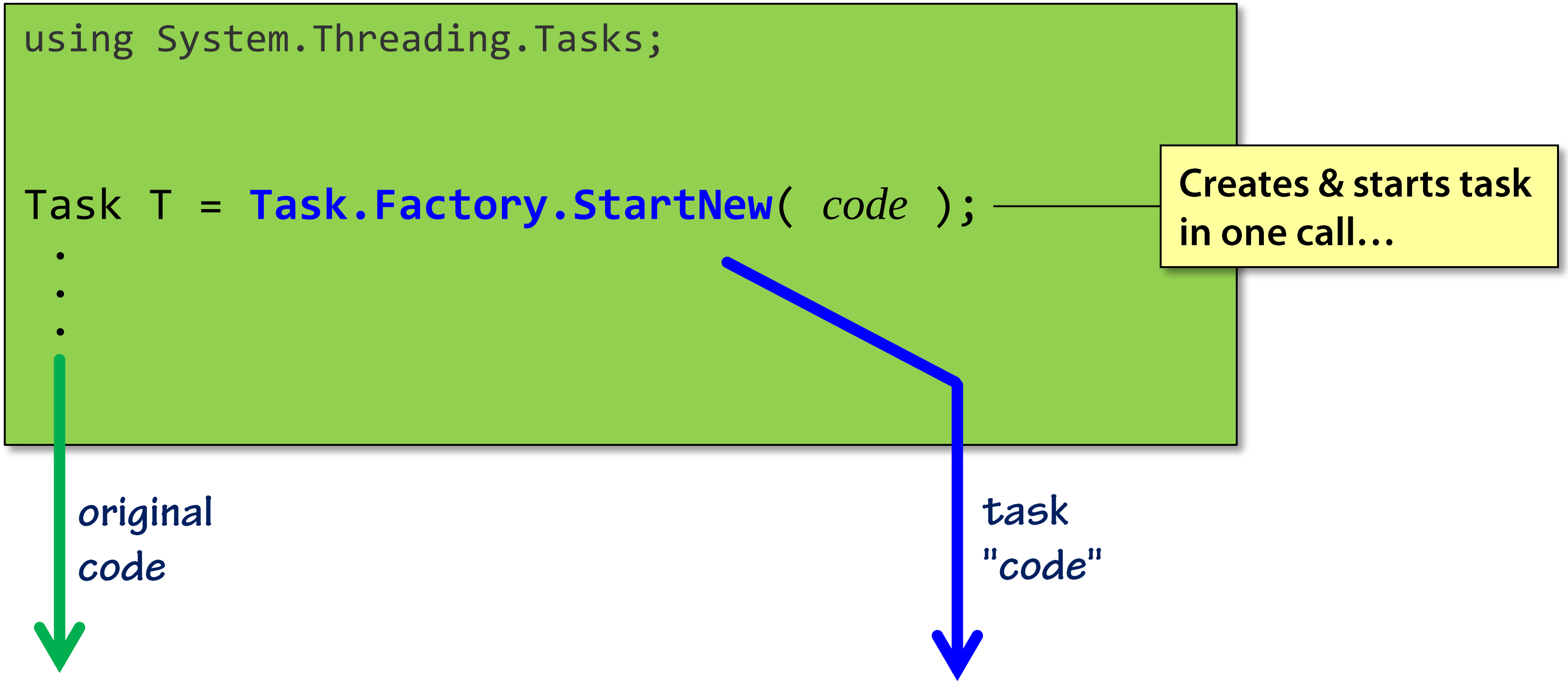
- Run the simulation as a separate task on a worker thread
- Update UI using a 2nd task running on UI thread

```
Task T = new Task( () =>
  {
    // Asian options simulation code:
  }
);
T.Start();
```

```
Task T2 = T.ContinueWith( (antecedent) =>
  {
    // code to update UI once simulation task is finished:
  },
  TaskScheduler.FromCurrentSynchronizationContext()
);
```

Creating code tasks — preferred approach

- Equivalent, but slightly more efficient…

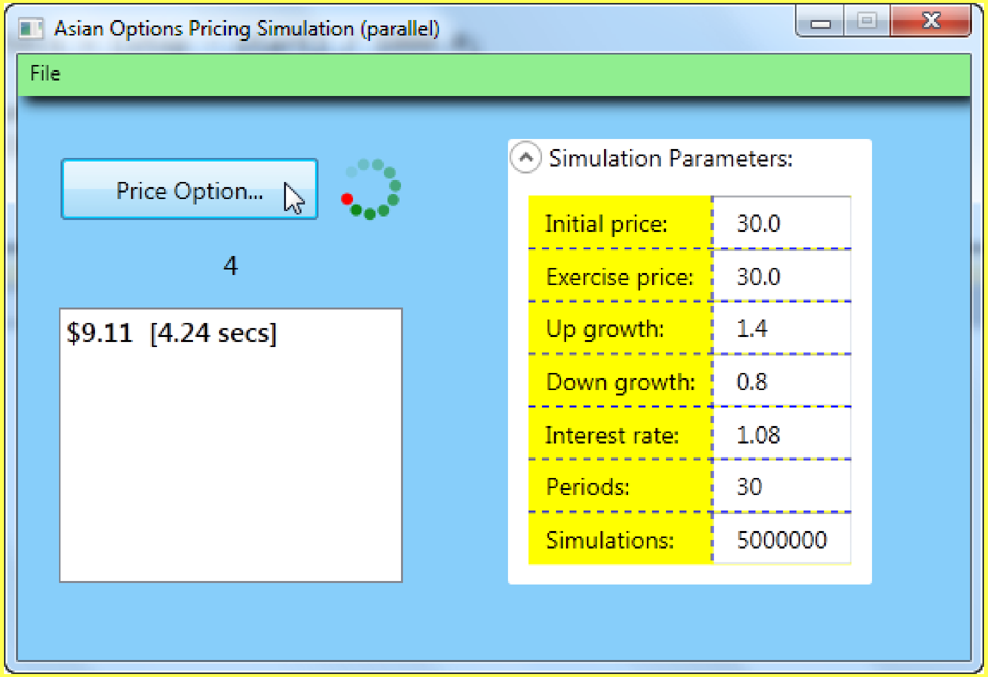


```
using System.Threading.Tasks;


Task T = Task.Factory.StartNew( code );
  .
  .
  .
```

Creates & starts task in one call…

DEMO

- Parallel programming for performance
- Asian Options finance modeling application, revisited…



$9.11 [4.24 secs]

Simulation Parameters:

Initial price:	30.0
Exercise price:	30.0
Up growth:	1.4
Down growth:	0.8
Interest rate:	1.08
Periods:	30
Simulations:	5000000

Language support

- **Task Parallel Library takes advantage of .NET language features**
- **In particular:**
  - Lambda expressions
  - Closures

Lambda expressions

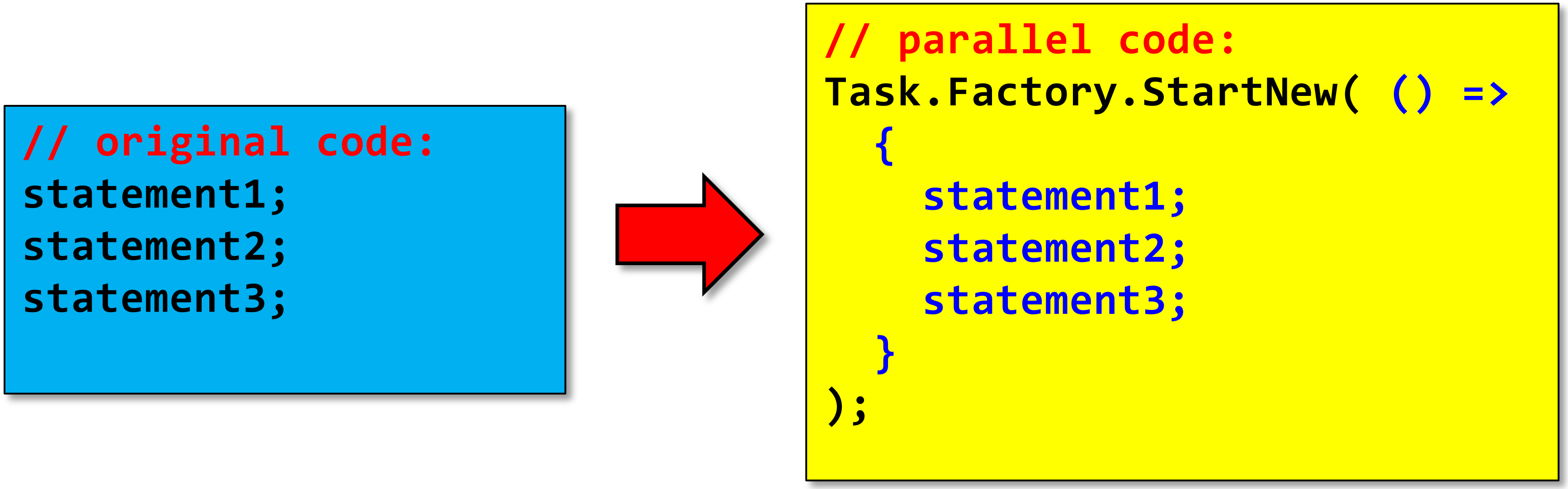
- **Lambda expression** == unnamed block of code

```
(parameters) =>
{
   code
}
```

key syntactic element identifying a lambda expression

Advantage

- TPL accepts lambda expressions as parameters
  - *making it easier to create tasks…*

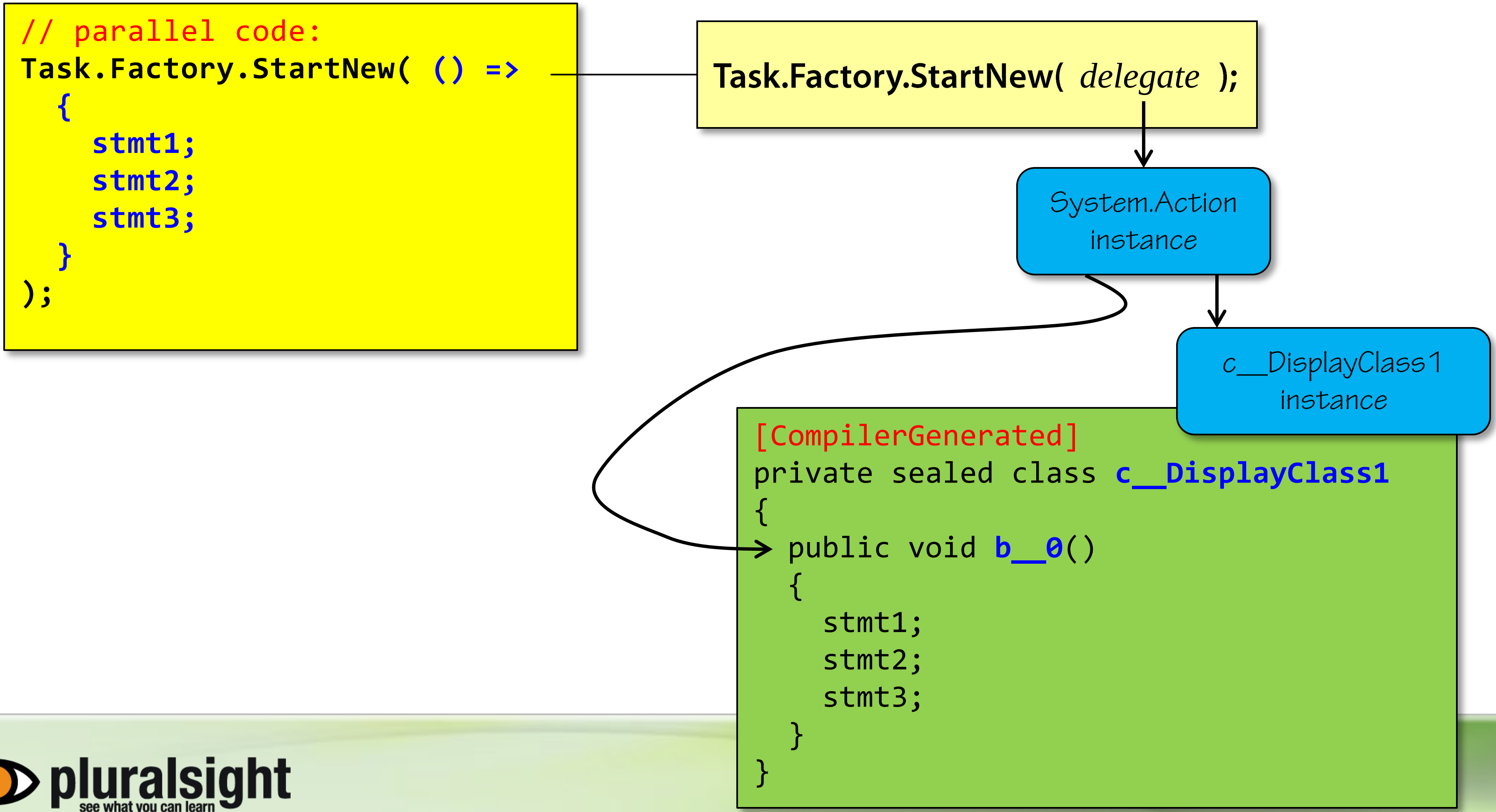


```
// original code:
statement1;
statement2;
statement3;
```

```
// parallel code:
Task.Factory.StartNew( () =>
  {
    statement1;
    statement2;
    statement3;
  }
);
```

Behind the scenes…

- Lambda expression == custom class + delegate



```
// parallel code:
Task.Factory.StartNew( () =>
  {
    stmt1;
    stmt2;
    stmt3;
  }
);
```

Task.Factory.StartNew(*delegate*);

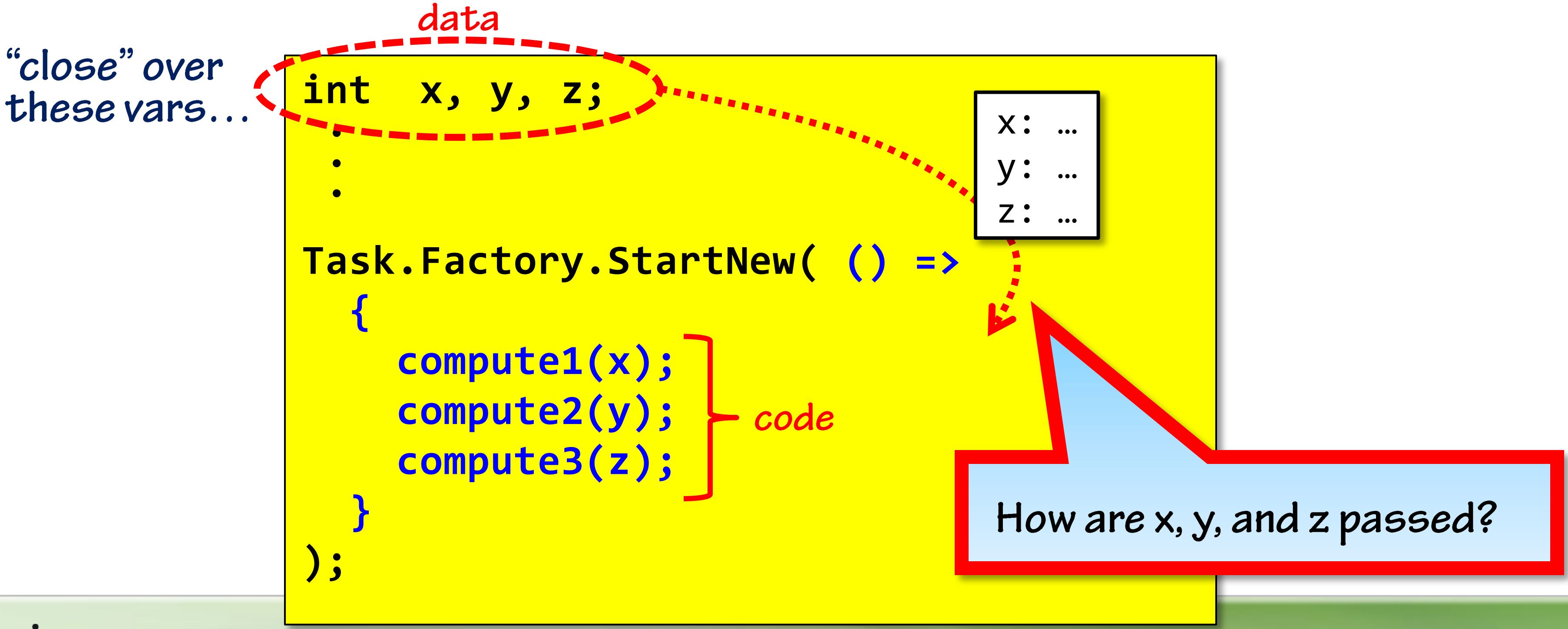
System.Action instance

c__DisplayClass1 instance

```
[CompilerGenerated]
private sealed class c__DisplayClass1
{
  public void b__0()
  {
    stmt1;
    stmt2;
    stmt3;
  }
}
```

Closures

- **Closure** == code + supporting data environment
- Compiler computes closure in response to lambda expression
  - *necessary for compilation — recall code lives in a compiler-generated class*
  - *incredibly convenient form of parameter passing*



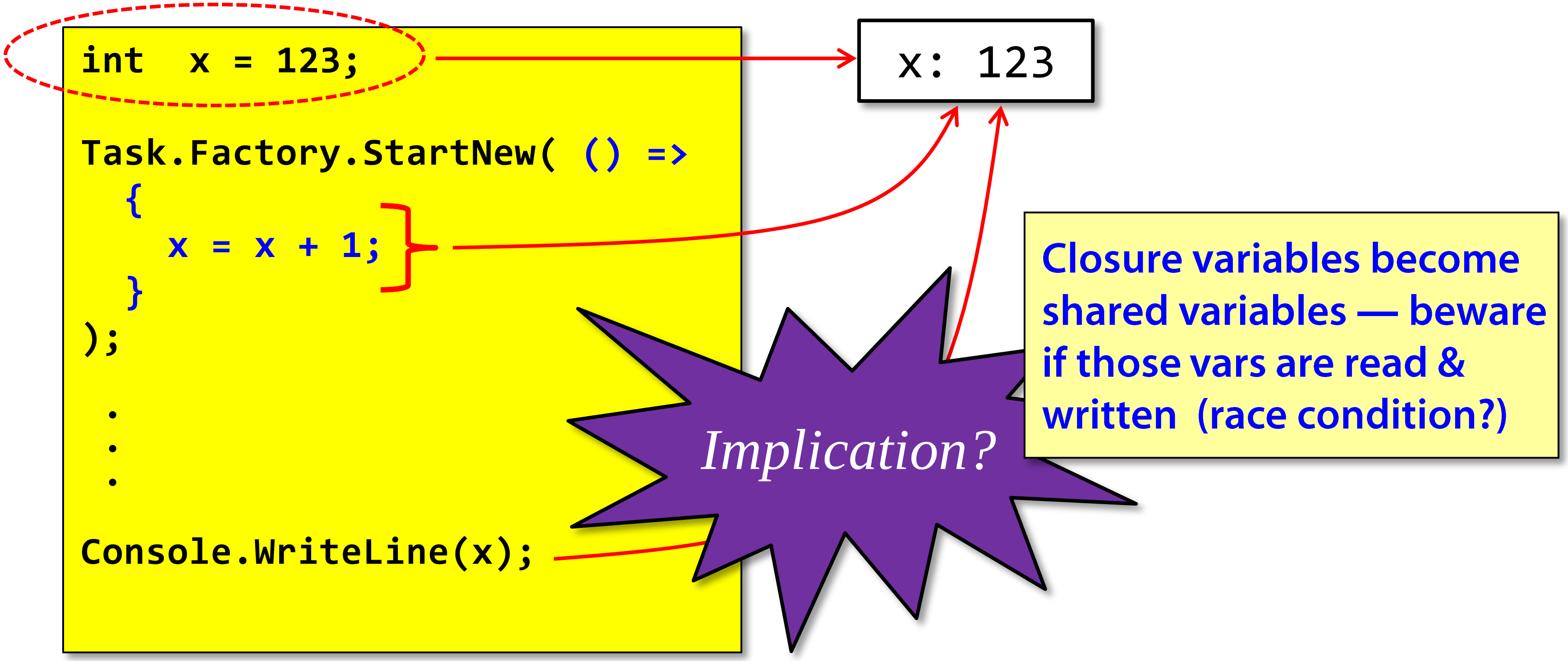
```
int  x, y, z;
  .
  .
  .
Task.Factory.StartNew( () =>
  {
    compute1(x);
    compute2(y);
    compute3(z);
  }
);
```

“close” over these vars…

How are x, y, and z passed?

Closure variables

- How are closure variables passed?
- Think "**BY REFERENCE**"

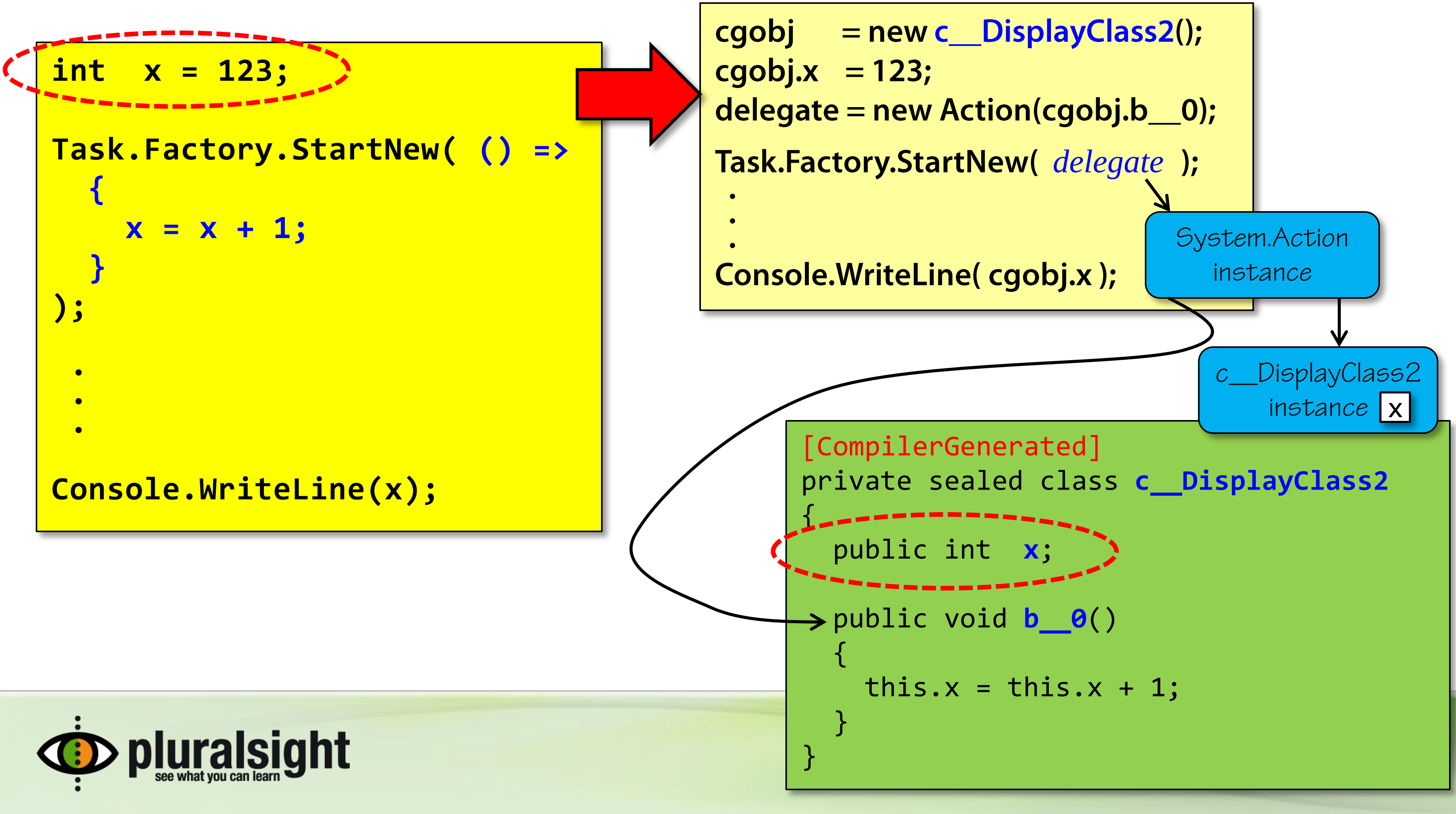


```
int  x = 123;

Task.Factory.StartNew( () =>
  {
    x = x + 1;
  }
);

 .
 .
 .

Console.WriteLine(x);
```

Closure variables become shared variables — beware if those vars are read & written (race condition?)

Behind the scenes (part 2)

- Closure variables are stored within compiler-generated class…



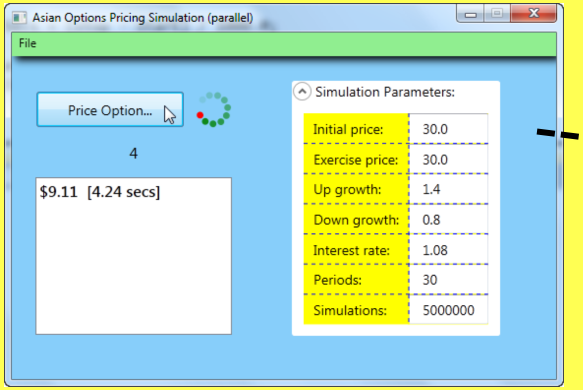
```
int  x = 123;

Task.Factory.StartNew( () =>
  {
    x = x + 1;
  }
);

  .
  .
  .

Console.WriteLine(x);
```

```
cgobj     = new c__DisplayClass2();
cgobj.x   = 123;
delegate = new Action(cgobj.b__0);
Task.Factory.StartNew( delegate );
 .
 .
 .
Console.WriteLine( cgobj.x );
```

```
[CompilerGenerated]
private sealed class c__DisplayClass2
{
  public int  x;

  public void b__0()
  {
    this.x = this.x + 1;
  }
}
```

DEMO

- Implementation details around lambda expressions
- Asian Options finance modeling application, revisited…

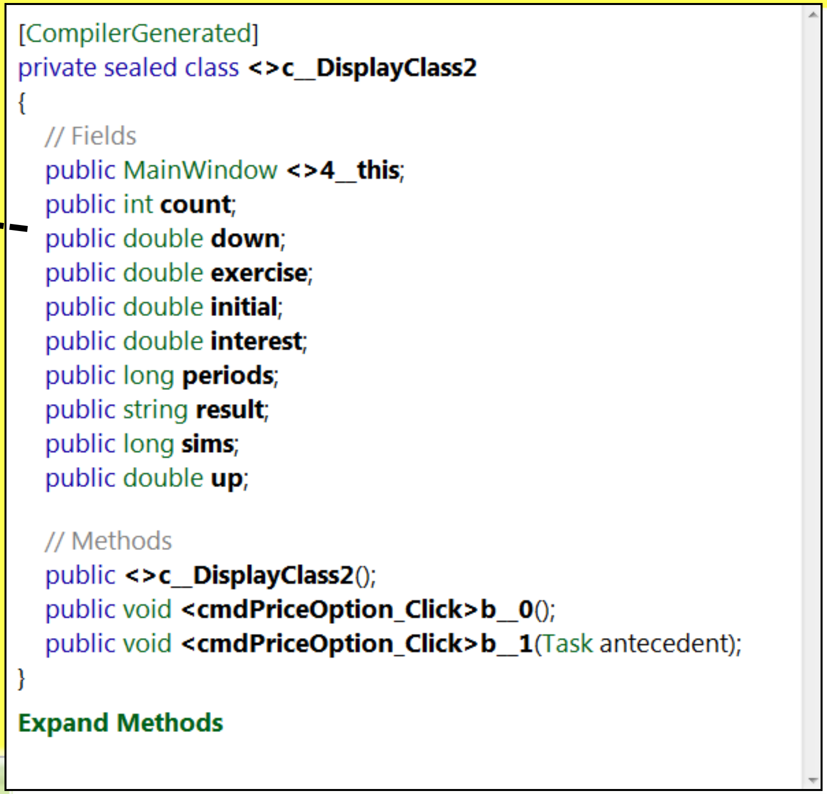


4

$9.11 [4.24 secs]

Simulation Parameters:

Initial price:	30.0
Exercise price:	30.0
Up growth:	1.4
Down growth:	0.8
Interest rate:	1.08
Periods:	30
Simulations:	5000000



```
[CompilerGenerated]
private sealed class <>c__DisplayClass2
{
    // Fields
    public MainWindow <>4__this;
    public int count;
    public double down;
    public double exercise;
    public double initial;
    public double interest;
    public long periods;
    public string result;
    public long sims;
    public double up;

    // Methods
    public <>c__DisplayClass2();
    public void <cmdPriceOption_Click>b__0();
    public void <cmdPriceOption_Click>b__1(Task antecedent);
}
```

Expand Methods

Types of tasks

- **You may have noticed we used the term code tasks**
- **Code tasks are *thready* — have explicit code, require thread to execute**
- **Are there other kinds of tasks?**

- **Yes!**
  - tasks can also be a **façade** over existing operations (e.g. asynchronous I/O)
  - code to execute is not explicitly provided, but implied elsewhere
  - such tasks can be *threadless* — e.g. HW is performing operation

```
var   existingOp = new TaskCompletionSource< ResultType >();
Task T_facade    = existingOp.Task;
  .
  .     // op now encapsulated as a task…
  .
```

```
// when op completes, let client know:
existingOp.SetResult(…);
```

Summary

- **Tasks are the new model for asynchronous & parallel programming**
  - **Async** ==> start an operation and then return to UI (responsiveness)
  - **Parallel** ==> divide workload so you finish sooner (performance)

- **Tasks denote a unit of work, an ongoing computation**
- **Responsibilities:**
  - **Your job** as a developer is to create tasks
  - **.NET's job** is to execute tasks as efficiently as possible

- **Beware:**
  - UI updates must be executed in context of UI thread
  - Closures yield the possibility of shared variables

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