Parallel and Asynchronous programming

.NET

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Contents

- Plain Threads
- Parallel vs Asynchronous
- TPL
- async / await

This module uses multithreading techniques to realize parallel and asynchronous tasks. First we introduce plain threads and explain why they are not enough to write complex programs. Then we introduce the concepts of “Parallel” and “Asynchronous” programming. The .NET framework introduced the **Task Parallel Library** (TPL) in .NET 4.0 to realize both parallel and asynchronous programming. Using the latest C# keywords **async** and **await**, the asynchronous part becomes much more readable. However this hides much of its complexity.

Plain Threads

- Class Thread accepts a delegate
- Methods for interacting with threads

```
public class Alpha
{
    // This method that will be called when the thread is started
    public void Beta()
    {
        while (true)
        {
            Console.WriteLine("Alpha.Beta is running in its own thread.");
            Thread.Sleep(500);
        }
    }
};
```

Demo: OnlyThreads

Reference: https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.110).aspx

Class Alpha defines a method that will be run by a separate thread.

Thread.Sleep simulates a long computation. Namespace: System.Threading

Working with Threads

```
Alpha oAlpha = new Alpha();

// Create the thread object, passing in the Alpha.Beta method
// via a ThreadStart delegate. This does not start the thread.
Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));

// Start the thread
oThread.Start();

// Spin for a while waiting for the started thread to become
// alive:
while (!oThread.IsAlive) ;

// Put the Main thread to sleep for 3 seconds to allow oThread
// to do some work:
Thread.Sleep(3000);
```

Demo: OnlyThreads

ThreadStart: delegate to create a Thread

- Start: start the thread
- IsAlive: checks thread
- Thread.Sleep: put current thread to sleep

Working with threads

```
// Request that oThread be stopped
oThread.Abort();

// Wait until oThread finishes. Join also has overloads
// that take a millisecond interval or a TimeSpan object.
oThread.Join();

Console.WriteLine();
Console.WriteLine("Alpha.Beta has finished");

try
{
    Console.WriteLine("Try to restart the Alpha.Beta thread");
    oThread.Start();
}
catch (ThreadStateException)
{
    Console.Write("ThreadStateException trying to restart Alpha.Beta. ");
    Console.WriteLine("Expected since aborted threads cannot be restarted.");
}
```

Demo: OnlyThreads

- Abort: aborts thread, you can't restart an aborted thread
- Join: wait for thread to finish

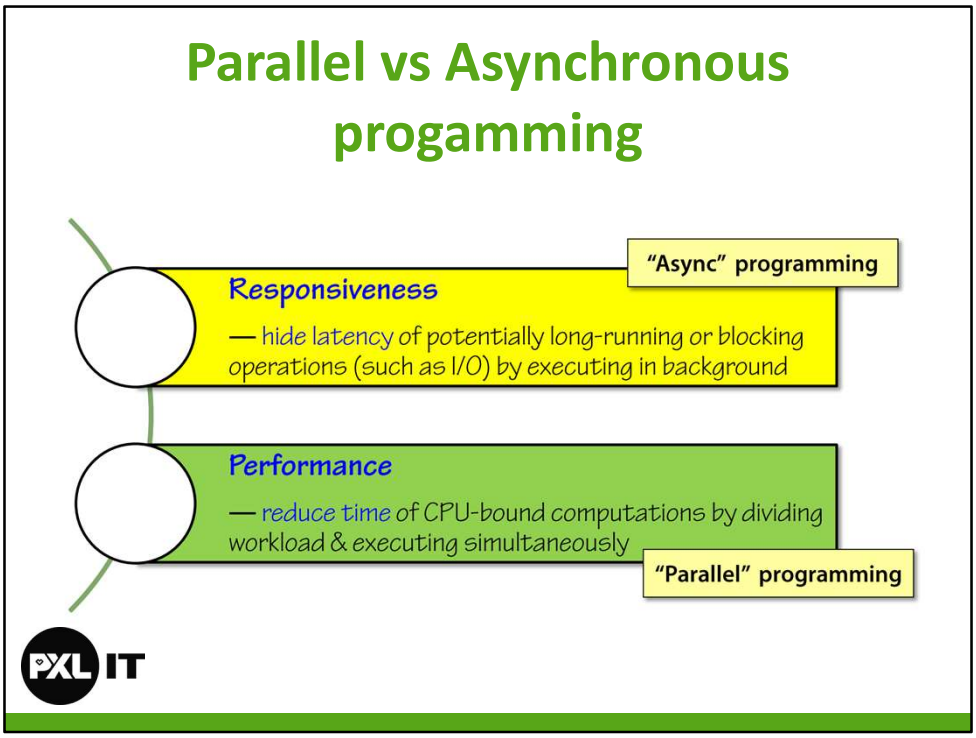
Threads are not enough

- A threads creates a separate virtual CPU inside your process (program)
- A thread is a resource, use it sparingly
- Multithreading is difficult:
  - Synchronization between threads
    - Deadlock
    - Race conditions
  - Thread pooling (resource sharing)
- Frameworks are created on top of threads

The concept of a Thread class is not enough to create complex programs. .NET has several frameworks to help manage the complexity and readability of your programs.

Deadlock: threads wait indefinitely for each other

Race condition: when sharing a variable, threads could possibly introduce errors



Responsiveness: you can still do other things in the UI while some other long or blocking operation is executed

Performance: to several things at once to get things done faster

Example Asynchronous

- Show a loading indicator
- Start an asynchronous operation
  - Read large CSV file from disk
  - Count the rows (one by one)
  - Signal that the asynchronous operation has completed
- Render the result on screen
- Remove the loading indicator

Suppose you need to load a long file from disk. How would you present this to the user?

Remark: while the program is loading, the user could still do other things, e.g. move or resize the window.

→ This example is NOT parallel programming! See the next slide.

Example Async and Parallel

- Show a loading indicator
- Start an asynchronous operation
  - Read large CSV file from disk
  - Process rows in parallel
  - Signal that the asynchronous operation has completed
- Render the result on screen
- Remove the loading indicator

PXL IT

Parallel programming is used to take a big problem and chop it up into smaller chunks that can be executed simultaneously.

Anology: boiling eggs

Parallel

- You can boil 10 eggs one by one for 10 minutes → 100 minutes or;
- You boil them all at once. This speeds up the process (all ready in 10 minutes)

Async

- If you set an egg timer, you could do something else
- The boiling process becomes asynchronous
- You will be notified when eggs are ready

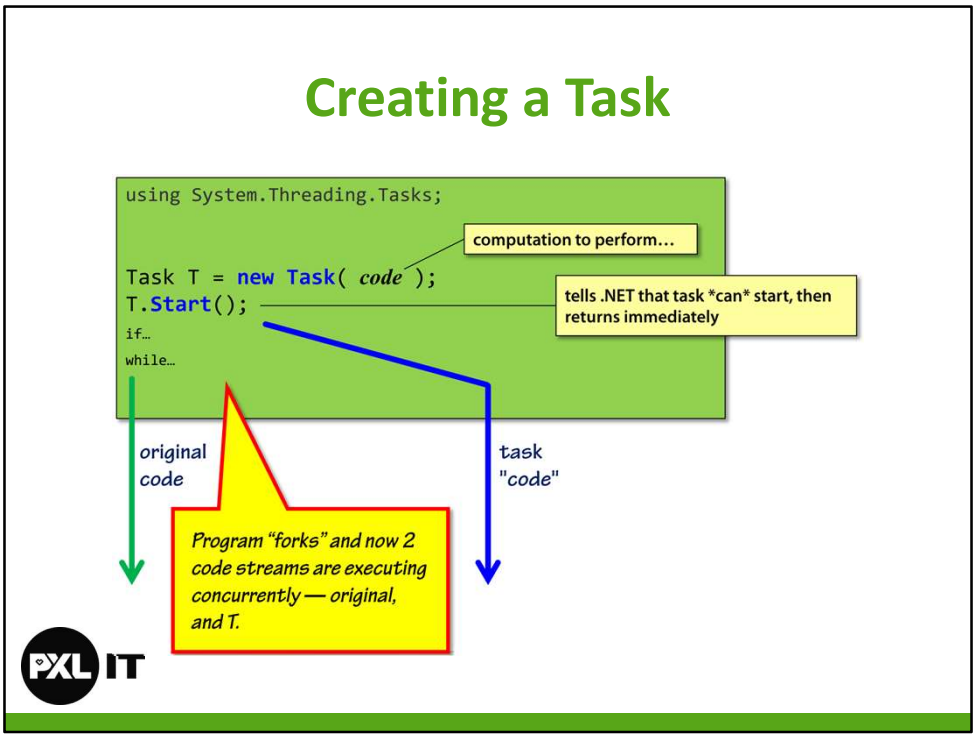
Task

- Definition
  - A unit of work
  - An object denoting an ongoing operation or computation

The Task Programming Library was introduced to allow to create parallel and asynchronous programs. It is build on top of plain threads and frees the developer of the complexities of working with threads alone.

Central concept: Task

https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task



The *code* parameter → delegate

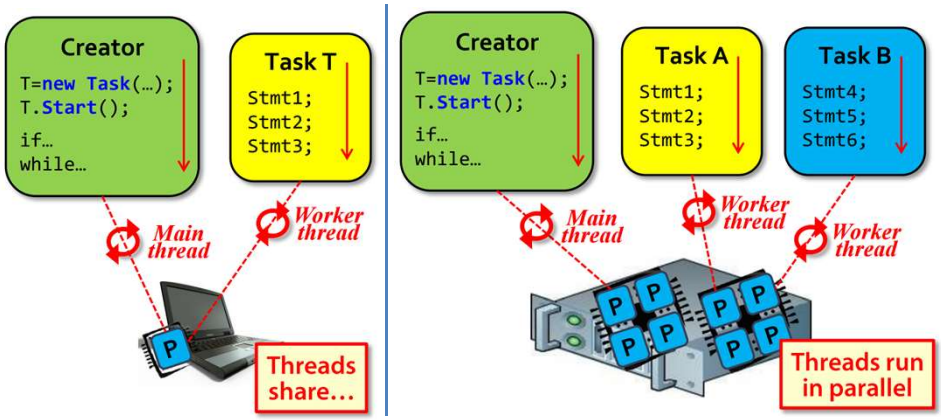
Start

- It *can* start
- Returns immediately
- Put into a queue for execution (so no direct Thread!)

The code forks into two execution paths

Execution model – quick look

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



Left: single core

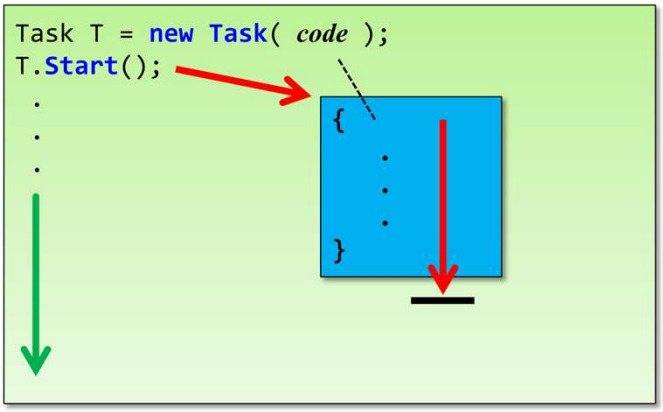
- Multiple threads are shared by the single core
- Context switching between threads does not seem useful
- But: the UI thread gets executed → responsiveness increases!

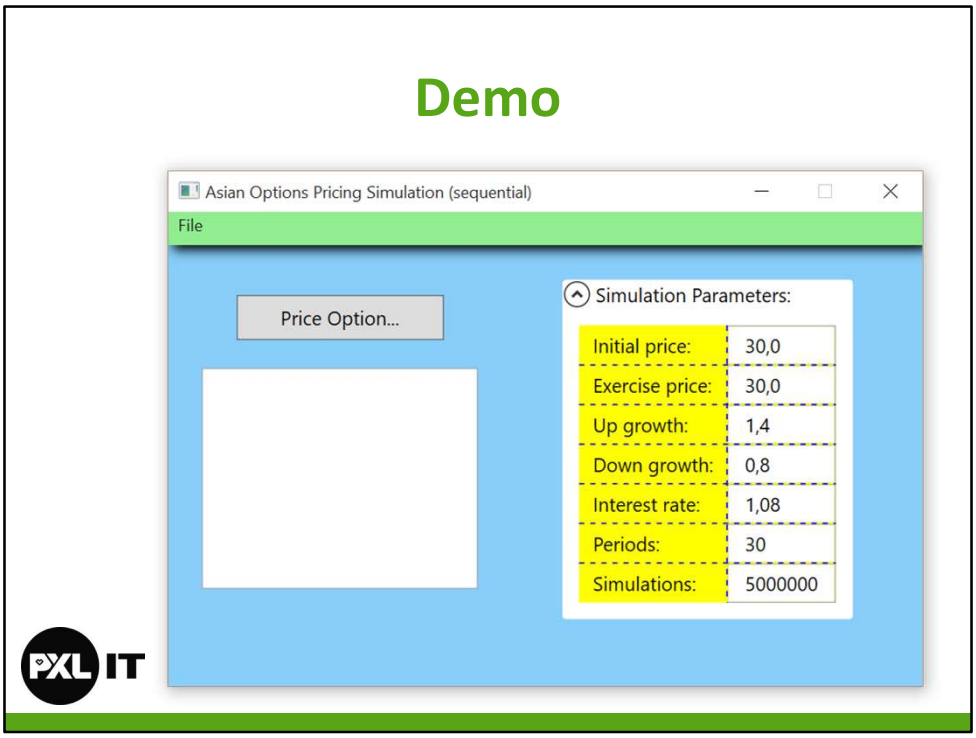
Right: multiple processors

- Tasks can really run in parallel

Task completion

- When does a task complete?
  - When the code block is exited – normally
  - Throwing an exception





Demo: AsyncOptions-Seq (and subsequent versions)

This demo is about asynchronous programming for **responsiveness**.

The application performs a large number of Monte Carlo simulations to model stock prices, in particular Asian options. In this demo, the app is parallelized at the coarse-grain level, i.e. each click of the "Simulation" button launches a task so the simulation can be done in parallel if a compute core is available. For example, on a quad-core machine, you can have as many as 4 simulations running in parallel.

The first goal is to make the app more responsive, so the UI does not freeze while a simulation is in progress.

Run the demo and notices that the program freezes while computation is going on.

There *is* code for a spinner, but this has no effect, since the computation is in the UI thread.

Note: the spinner itself is in a assembly called "Fenestra" and comes from this article:

http://blogs.msdn.com/b/pigscanfly/archive/2010/01/01/bizzyspinner-a-wpf-spinning-busy-sate-indicator-with-source.aspx

Production quality spinners can be found here:

http://mahapps.com/controls/progress-ring.html

https://github.com/100GPing100/LoadingIndicators.WPF

Demo Summary

- Run the simulation as a separate Task
- Update UI in context of UI thread

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

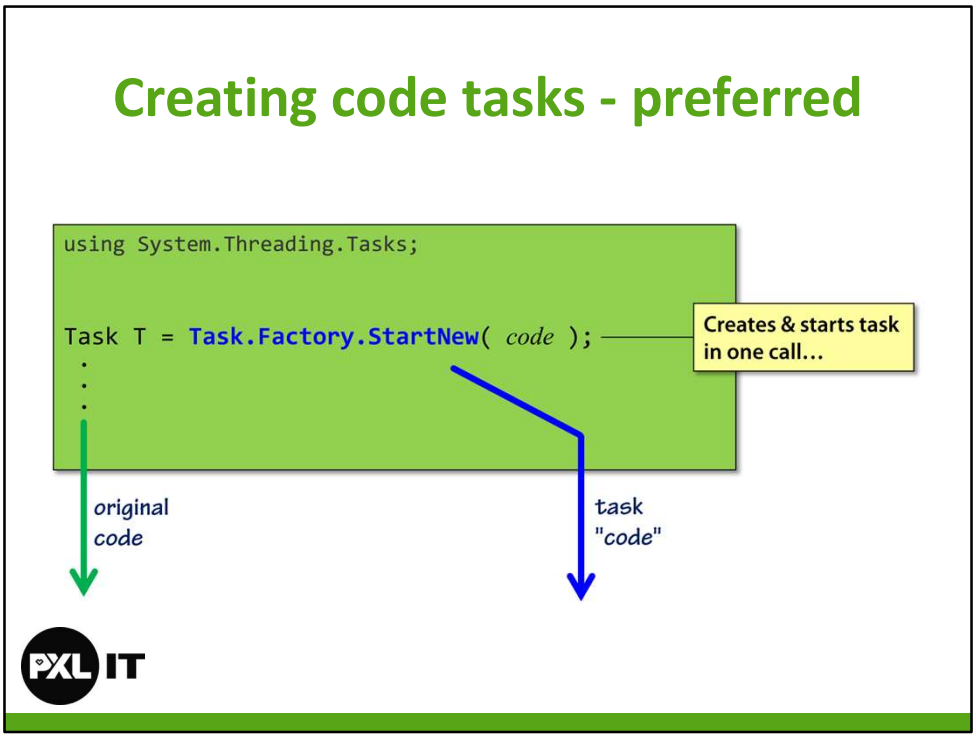
```
Task T2 = T.ContinueWith( (antecedent) =>
  {
    // code to update UI once simulation task is finished:
  },
  TaskScheduler.FromCurrentSynchronizationContext()
);
```

Demo: AsianOptions-Step3

T.ContinueWith

Wacht tot T gedaan is en ga verder met volgende taak => parameter antecedent

TaskScheduler.FromCurrentSynchronizationContext => UI thread

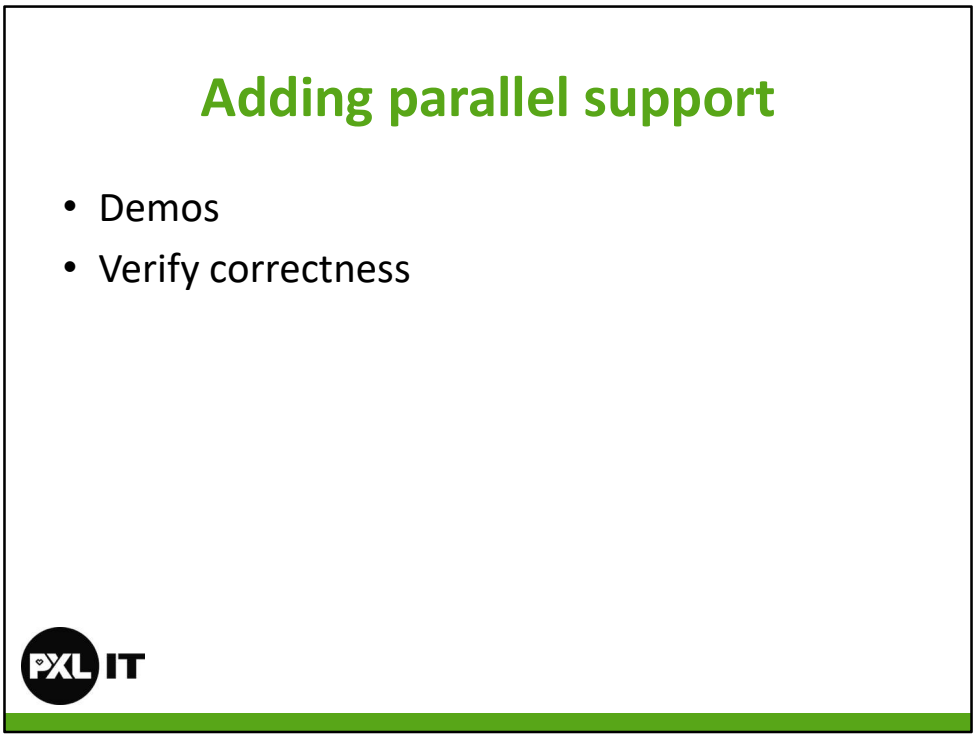


Task.Factory.StartNew

Create and Start in 1 call => more efficient

Nowadays, it is more common to do:

Task.Run



Demo: AsianOptions-Step4 en AsianOptions-Step5

Because variable m_counter is incremented en decremented in the same UI thread, there is no race condition.

Language support

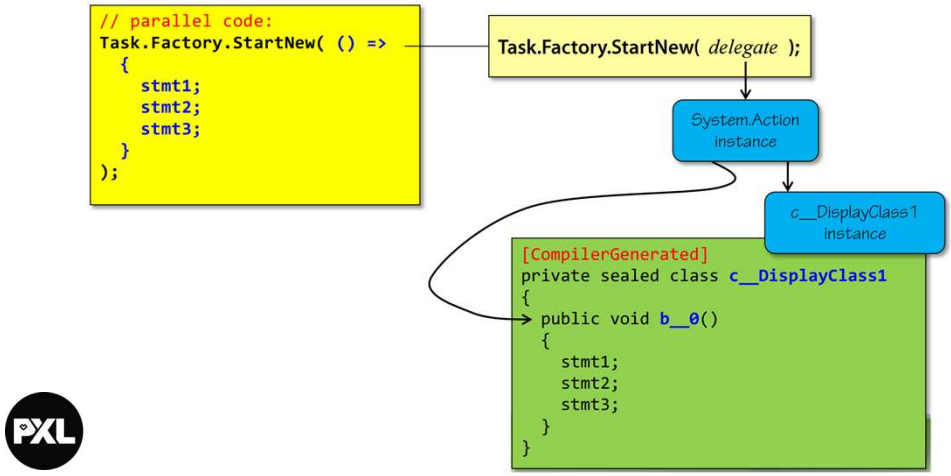
- Task Parallel Library (TPL) takes support of .NET language features:
  - Lambda expressions
  - Closures

Lambda expressions → already known

In orde to explain Closures, we need to know how lambda expressions are compiled to IL

Behind the scenes

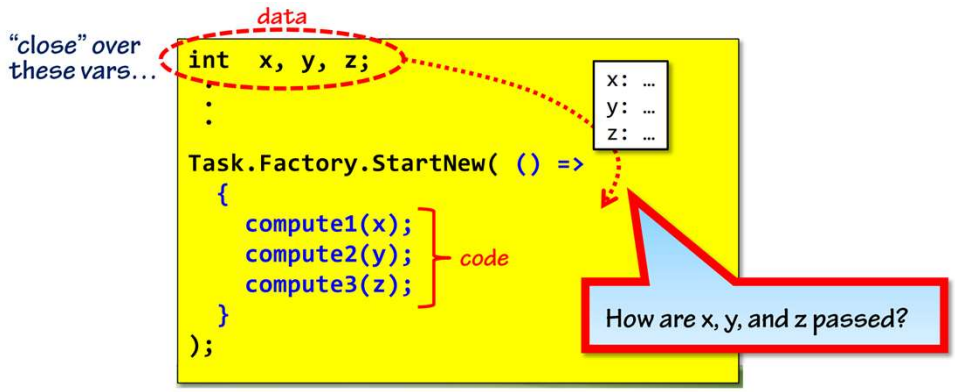
- Lambda expression == custom class + delegate



For each delegate, there is an Action instance which has a custom delegate that points to a method in a (generated) class.

Closures

- **Closure** == code + supporting data environment
- Compiler computes closure in response to lambda expression



How are x, y, z passed into the method body of the closure?

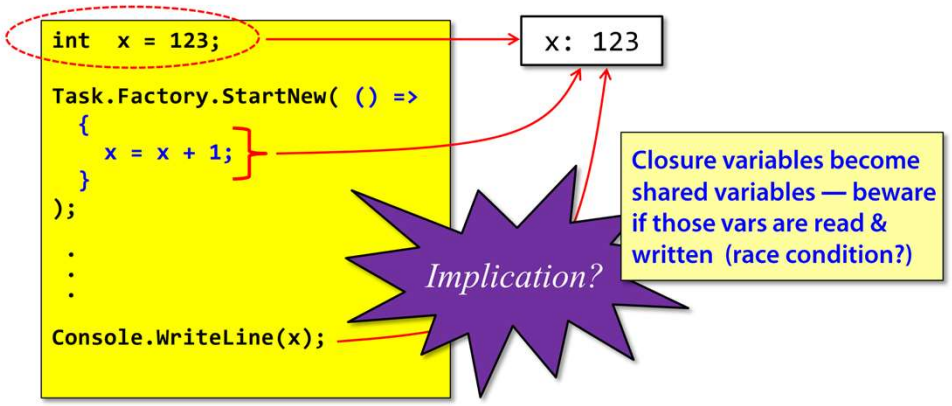
Compiler generates the necessary code to pass these in **by reference**, so x, y, z inside the lambda point to the same variables as x, y and z (this can be verified with ILSpy).

Note: even if the types are value types (int, struct, etc) !!

This is very handy, because you don't have to maintain parameters for methods. You simply know that the surrounding context from outside the lambda is available inside.

Closure variables

- Passed **by reference**



x is passed by reference into the lambda

This is a task → executes on a different thread

So what is the output? That depends on who is first → race condition.