Module 01:
"Threads"

Agenda

▸ **Introduction to Threads**

▸ Threads and Data

▸ Controlling Threads

▸ Exception Handling

▸ Debugging Tips 'n Tricks

Hello World ☺

```csharp
static void Main()
{

    Thread thread = new Thread(WriteWorld);
    thread.Start();


    for (int i = 0; i < 1000; i++)
    {
        Console.Write("Hello ");
    }

}
```

```csharp
static void WriteWorld()
{

    for (int i = 0; i < 1000; i++)
    {
        Console.Write("World ");
    }

}
```

Managing Threads

- Thread Scheduler manages threads
  - CLR
  - Operating System

- Single processor ~ Time-slicing
- Multiple processors ~ Concurrency + Time-slicing

- Preemption = Thread execution is interrupted

- Threads vs. Processes

Use Cases for Threads

▸ Responsive UI
▸ Simultaneous processing of requests or updates
▸ Parallel Programming
▸ Efficient use of CPUs
▸ Speculative Execution

▸ But... Threads incur complexity issues!

Important Thread Properties

▸ Always a "executing" thread
- `Thread.CurrentThread`

▸ Each Thread has a number of important properties
- `ManagedThreadId`
- `IsAlive`
- `IsBackground`
- …

Foreground vs. Background

▸ Foreground threads
- Keep the application running

▸ Background threads
- Shut down silently when application closes
- Note: finally and using blocks do not always complete!

▸ No difference in priority, scheduling, etc...

Agenda

▸ Introduction to Threads

▸ **Threads and Data**

▸ Controlling Threads

▸ Exception Handling

▸ Debugging Tips 'n Tricks

Sharing Data Between Threads

▸ Each thread has a
- Separate call stack
- Separate local variables

▸ Shared between threads
- Static members
- Object state

▸ Well... Unless...

ThreadStatic and ThreadLocal<T>

▸ Default data sharing between threads can be manually changed

▸ **ThreadStatic** attribute
- Enforces per-thread copy of static data
- Note: This is non-obvious! Don't use this!

▸ **ThreadLocal<T>** class
- Lazily created, per-thread local variable
- Introduced in .NET 4.0

Starting Threads

▸ **ThreadStart** delegate is implicit (since C# 2.0)

```
public delegate void ThreadStart();
```

```
Thread t1 = new Thread( new ThreadStart (a.Go) );
```

▸ Can also use lambda expressions etc.

```
Thread t3 = new Thread( () =>
{
    int counter = 0;
    for (int i = 0; i < 100; i++)
    {
        counter++;
    }
});
```

Passing Data to Threads

▸ **ParameterizedThreadStart** delegate is

```
public delegate void ParameterizedThreadStart (object obj);
```

```
Thread t1 = new Thread( new ParameterizedThreadStart (a.Go) );
```

▸ This value is passed using an overload of **Start()**.

```
t1.Start(100);
```

```
void Go(object input)
{
    int max = (int)input; // Needs ugly cast
    ...
}
```

Capturing Data in Threads

▸ Data can also be *captured* by threads via lambdas

▸ Beautiful, elegant, powerful, and...

▸ ... a very common source of hard-to-spot errors!

```
string text = "Hello, World";
Thread t1 = new Thread(() => Console.WriteLine(text));

text = "WTF?!?";
Thread t2 = new Thread(() => Console.WriteLine(text));

t1.Start();
t2.Start();
```

Agenda

▸ Introduction to Threads
▸ Threads and Data
▸ **Controlling Threads**
▸ Exception Handling
▸ Debugging Tips 'n Tricks

Blocking vs. Spinning

▸ A thread can be blocked
- **Sleep()** ～ Pause for a time period
- **Yield()** ～ Almost a **Sleep(0)**

```
Thread.Sleep( 1000 );
Thread.Yield();
```

- Usually not for production code

▸ A thread can spin
- **SpinWait()** ～ Busy-wait for a number of cycles

```
Thread.SpinWait( 100_000_000 );
```

- Caution! Only for very advanced scenarios

Join

- ▸ You can wait for a thread to end using Thread.Join()

```
Thread t = new Thread(Go);
t.Start();
t.Join(); // Wait for t to end
Console.WriteLine("t has completed!");
```

- ▸ Note: The calling thread is blocked during Join()

- ▸ There is an overload with timeout period specified

```
if( t.Join(5000) )
{
    Console.WriteLine("t has completed within 5 seconds!");
}
```

Interrupt

▸ A blocked thread can be *interrupted*

```
Thread t = new Thread(Go);
t.Start();
...;
t.Interrupt();
```

▸ A `ThreadInterruptedException` is thrown at blocking location
- Must be handled by thread method
- Exception is not re-thrown if unhandled

▸ Note: You'll probably never need Thread.Interrupt().

▸ We will encounter better alternatives for signalling later... ☺

Abort

▸ A blocked thread can be forcibly released by *aborting*

```
Thread t = new Thread(Go);
t.Start();
...;
t.Abort();
```

▸ A `ThreadAbortException` is thrown at immediate location
- Unpredictable! "May abort thread"
- Exception is re-thrown if unhandled

▸ Note: Don't use Thread.Abort()
- Most framework code is not safe to abort

Thread Priority

▸ Thread priority <u>within</u> the process can be set

```
Thread t = new Thread(Go)
{
    Priority = ThreadPriority.Highest;
}
t.Start();
```

```
enum ThreadPriority
{
    Lowest = 0,
    BelowNormal = 1,
    Normal = 2,
    AboveNormal = 3,
    Highest = 4
}
```

▸ Don't be fooled... by "everything"!

▸ Process priority is more important

```
using (Process p = Process.GetCurrentProcess())
{
    p.PriorityClass = ProcessPriorityClass.High;
}
```

▸ OS is allowed to ignore thread priority

Rule of Thumb

Q: "For large problem sizes, what is the preferred amount of threads to use?

A: Usual rule of thumb is
Two threads per logical core

Agenda

▸ Introduction to Threads
▸ Threads and Data
▸ Controlling Threads
▸ Exception Handling
▸ Debugging Tips 'n Tricks

Don't...

▸ Never handle exceptions at thread creation or start!

```
try
{
    Thread t = new Thread(() =>
    {
        ...
        throw new InvalidOperationException("Argh!");
    });
    t.Start();
}
catch (Exception)
{
    Console.WriteLine("Thread error");
}
```

Do...

▸ Always handle exceptions in thread method!

```
Thread t = new Thread(Go);
t.Start();
```

```
void Go()
{
    try
    {
        ...
        throw new InvalidOperationException("Argh!");
    }
    catch (Exception)
    {
        Console.WriteLine("Thread error");
    }
}
```

C# 6.0 Exception Filters

▸ Exception filters facilitates the handling of exceptions matching a specific type and/or predicate

```
var from = Bank.CreateAccount(100);
var to = Bank.CreateAccount(100);

try
{

    Bank.TransferFunds(from, 200, to);
}
catch (InsufficientFundsException e) when (e.Account?.IsVIP == true)
{
    // Handle VIP account
}
```

▸ Distinct clauses can match same exception type but with different conditions

Unhandled Exceptions

▶ Remember: Exceptions unhandled in threads will terminate process

▶ `.AppDomain.CurrentDomain` offers a handler to catch all exceptions
  - But does not prevent shutdown

```
AppDomain.CurrentDomain.UnhandledException += OnException;


Thread t = new Thread(Go);
t.Start();
```

```
void OnException(object sender, UnhandledExceptionEventArgs e)
{
    ...
}
```
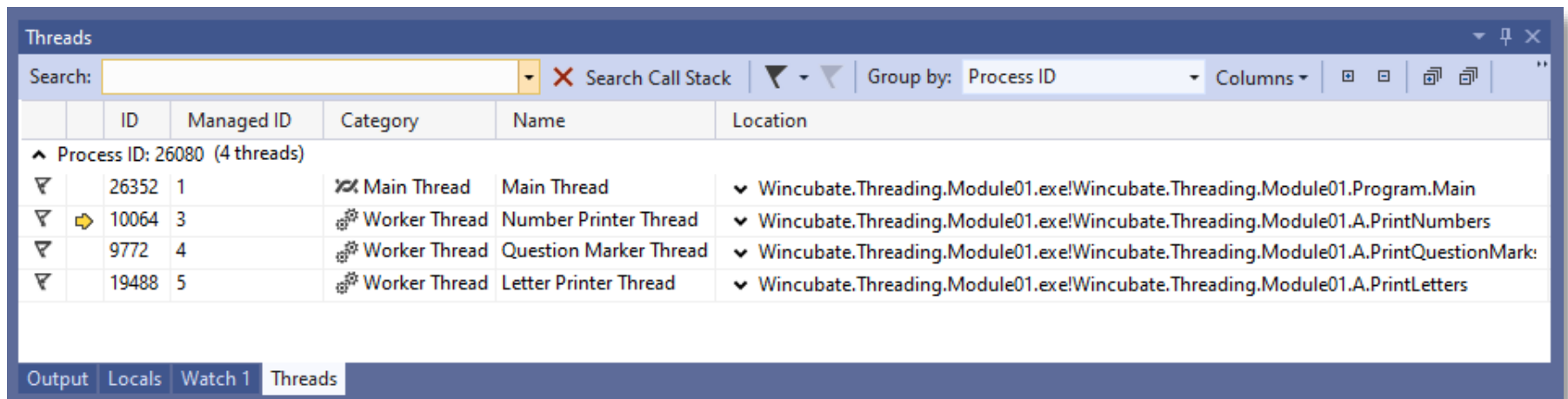
Agenda

▸ Introduction to Threads
▸ Threads and Data
▸ Controlling Threads
▸ Exception Handling
▸ **Debugging Tips 'n Tricks**

Debugging Threads Tips 'n Tricks

▸ Visual Studio Threads Window
  - Debug > Windows > Threads (Ctrl+D, T)

		ID	Managed ID	Category	Name	Location
Threads	Search:				▾ ✕ Search Call Stack ▼ ▾ ▼	Group by: Process ID ▾ Columns ▾ ⊞ ⊟ ⊡ ⊡
∧ Process ID: 26080 (4 threads)						
▽		26352	1	✕✕ Main Thread	Main Thread	⌄ Wincubate.Threading.Module01.exe!Wincubate.Threading.Module01.Program.Main
▽	⇨	10064	3	⚙ Worker Thread	Number Printer Thread	⌄ Wincubate.Threading.Module01.exe!Wincubate.Threading.Module01.A.PrintNumbers
▽		9772	4	⚙ Worker Thread	Question Marker Thread	⌄ Wincubate.Threading.Module01.exe!Wincubate.Threading.Module01.A.PrintQuestionMark
▽		19488	5	⚙ Worker Thread	Letter Printer Thread	⌄ Wincubate.Threading.Module01.exe!Wincubate.Threading.Module01.A.PrintLetters

Output | Locals | Watch 1 | Threads

▸ `Thread.Name` property helps debugging

Summary

▸ Introduction to Threads

▸ Threads and Data

▸ Controlling Threads

▸ Exception Handling

▸ Debugging Tips 'n Tricks

WINCUBATE

Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net

Ringgårdsvej 4A
8270 Højbjerg
Denmark