Module 13

"Asynchronous Programming"

TEKNOLOGISK INSTITUT

TEKNOLOGISK
INSTITUT

Agenda

- Introducing Task Parallel Library
- Combining and Handling Tasks
- Threading Issues

TEKNOLOGISK
INSTITUT

Task Parallel Library

- Task Parallel Library (TPL)
  - Was introduced in .NET 4.0
  - Enhanced in .NET 4.5
    - Special keywords are included in C# 5.0
- Features
  - Task Parallelism
  - Data Parallelism
  - Parallel LINQ
  - Thread-safe collections
- Emerging trends leverage parallelism! Also .NET!

TEKNOLOGISK
INSTITUT

Creating Tasks

- The Task class captures a unit of computation
- Initialized from constructor using a computation described by
  - Action delegate
  - Anonymous method
  - Lambda expression (usually preferred)

```
Task task = new Task( () =>
   Console.WriteLine( "Hello World from Task Parallel Library" )
);
```

- Note: Does not run automatically when created!

TEKNOLOGISK INSTITUT

Task Execution

- Three approaches to starting tasks
  - Create `Task` object and invoke `Task.Start()`
  - Use `Task.Factory.StartNew()` static
  - Use `Task.Run()` static

```
Task task = Task.Factory.StartNew( () =>
{
   for ( int i = 1 ; i < 100 ; i += 2 )
   {
      Console.WriteLine( "\t" + i );
   }
});
```

- Usually one of the last two options is employed

TEKNOLOGISK INSTITUT

Waiting for Task Completion

- Tasks can be awaited
  - `Task.Wait()`
  - `Task.WaitAny()` static
  - `Task.WaitAll()` static

```
Task task1 = ...;
Task task2 = ...;
Task task3 = ...;

task1.Wait();

Task.WaitAny( task1, task2, task3 );

Task.WaitAll( task1, task2, task3 );
```

Tasks with Results

TEKNOLOGISK
INSTITUT

- `Task<T>`
  - captures a task returning a result of type `T`
- `Task.Run<T>()` and `Task.StartNew<T>()` also exist

```
Task<DateTime> t = Task.Run<DateTime>( () => DateTime.Now );

Console.WriteLine( t.Result );
```

- Result can be explicitly retrieved via `Task.Result`
  - Note: This property is blocks when task is not yet completed!

TEKNOLOGISK INSTITUT

Cancelling Tasks

- Running tasks can be requested cancelled
  - Signal token created by **`CancellationTokenSource`** class
  - Other code signal token supplied to task
- Task method then
  - Checks if cancellation is requested
  - Throws **`OperationCanceledException`** to accept cancellation

```
task = Task.Factory.StartNew( () =>
{  ...
   if( token.IsCancellationRequested )
   {
      throw new OperationCanceledException( token );
   }
}
```

- Check task running status via **`Task.Status`**

The `Parallel` Class

TEKNOLOGISK INSTITUT

- The `Parallel` class leverages data parallelism

- `Parallel.`
  - `Invoke()` invokes actions in parallel
  - `For()` is a parallel `for`-loop
  - `ForEach()` is a parallel `foreach`-loop

```
Parallel.For( 0, 1000, i =>
   Console.WriteLine(
     string.Format( "Executing number {0,4}...", i )
   )
);
```

- Developer's responsibility that iterations are in fact independent

Parallel LINQ

TEKNOLOGISK INSTITUT

- PLINQ = Parallel LINQ
  - **`ParallelEnumerable`** class is defined in **`System.Linq`** namespace

- `ParallelEnumerable`
  - `AsParallel<T>()`
  - `AsSequential<T>()`
  - `WithCancellation<T>()`
  - `WithDegreeOfParallelism<T>()`

```
var even = from i in numbers.AsParallel()
                            .WithCancellation()
                            .WithDegreeOfParallelism( 2 )
           where i % 2 == 0
           select i;
```

TEKNOLOGISK
INSTITUT

Agenda

- Introducing Task Parallel Library
- **Combining and Handling Tasks**
- Threading Issues

Combining Tasks

TEKNOLOGISK INSTITUT

- Tasks can be combined using `Task.ContinueWith()`

```
Task<DateTime> t1 = new Task<DateTime>( () =>
   DateTime.Now );
Task<string> t2 = t1.ContinueWith( previous =>
   string.Format("The time is {0}!", previous.Result ) );

t1.Start();
Console.WriteLine( t2.Result );
```

- Combinators include
  - `Task.WhenAll()`
  - `Task.WhenAny()`
  - `Task.Delay()`

- `TaskCreationOptions` allows the creation of child tasks

Task Exceptions

TEKNOLOGISK INSTITUT

- Task exceptions are thrown when
  - Waiting for task
  - Getting result for task
- `AggregateException` instances are thrown
  - Consists of a number of inner exceptions

```
try
{
   t.Wait();
}
catch ( AggregateException ae )
{
   foreach( Exception e in ae.InnerExceptions )
   {
      Console.WriteLine( e.Message );
   }
}
```

C# 5.0 **`await`** Operator

TEKNOLOGISK INSTITUT

- C# 5.0 introduces **`await`** keyword for methods returning `Task` or `Task<T>`
  - Yields control until awaited task completes
  - Results gets returned

- Allows you to program just like for synchronous programming…!

```
WebClient client = new WebClient();
string result = await client.DownloadStringTaskAsync( ... );

Console.WriteLine( result );
```

- Really complex control flow under the hood is made stunningly simple by compiler

C# 5.0 **async** Modifier

TEKNOLOGISK INSTITUT

- C# 5.0 introduces **async** keyword
  - Marks method or lambda as asynchronous
  - Note: Methods making use of **await** must be marked "**async**"

- You can now easily define your own asynchronous methods

```
async static void DoStuff()
{
   // ...

   string result = await client.DownloadStringTaskAsync( ... );

   // ...
}
```

- Can create async methods returning **void**, **Task**, or **Task<T>**

Exceptions Thrown by Tasks and Awaitable Methods

TEKNOLOGISK INSTITUT

- Observe and catch exceptions "as usual" when awaiting tasks

```
try
{
   string data = await client.DownloadStringTaskAsync( ... );
}
catch ( WebException ex ) { ... }
```

- Subscribe to unobserved exceptions through the `TaskScheduler.UnobservedTaskException` event

```
TaskScheduler.UnobservedTaskException +=
   ( object s, UnobservedTaskExceptionEventArgs ute ) => {
      foreach( Exception e in ute.Exception.InnerExceptions )
      {
          ...
      };
```

Tasks and Asynchronous Programming Model

TEKNOLOGISK INSTITUT

- The "traditional" .NET Asynchronous Programming Model consists of
  - `Begin`*`Xxx`*`()` methods
  - `End`*`Xxx`*`()` methods

- Tasks encapsulate this model using `TaskFactory.FromAsync()`

```
HttpWebResponse response =
   await Task<WebResponse>.Factory.FromAsync(
      request.BeginGetResponse,
      request.EndGetResponse,
      request )
   as HttpWebResponse;
```

TEKNOLOGISK
INSTITUT

Agenda

- Introducing Task Parallel Library
- Combining and Handling Tasks
- **Threading Issues**

TEKNOLOGISK
INSTITUT

Synchronizing Tasks

- Processor and operating system schedule tasks in and out repeatedly
  - Thread context switch can occur at any time
    - Even in the middle of assignments and increments etc.
- Hence computations need to be computationally safe
  - Some operations must be performed indivisibly!
  - Race conditions should be avoided
- Basically two solutions
  - Synchronizing access to critical regions of code
  - Signaling between threads

The `Monitor` Class

TEKNOLOGISK INSTITUT

- The `Monitor` class is a light-weight mechanism for use within a single process
  - `Monitor.Enter` static
  - `Monitor.TryEnter` static
  - `Monitor.Exit` static
- The `lock` keyword in C# is based on `Monitor` and `try-finally`

```
object syncObject = new object();
...
lock( syncObject )
{
   _counter++;
}
```

- Exam tip:
  - Lock can only lock reference types…!

Concurrent Collections

TEKNOLOGISK INSTITUT

- Thread-safe collection alternatives are provided in the `System.Collections.Concurrent` namespace
  - `ConcurrentQueue<T>`
  - `ConcurrentStack<T>`
  - `ConcurrentDictionary<K,V>`
  - `ConcurrentBag<T>`
  - `BlockingCollection<T>`

- Implement your own concurrent collection using
  - `IProducerConsumerCollection<T>`

Quiz: Asynchronous Programming Right or Wrong?

TEKNOLOGISK INSTITUT

```
await Console.WriteLine( "Hello, World" );
```

```
WebClient client = new WebClient();
await client.DownloadFile(
   "http://www.wincubate.net/BusinessCard.jpg"
);
```

```
WebClient client = new WebClient();
await client.DownloadFileTaskAsync(
   "http://www.wincubate.net/BusinessCard.jpg"
);
```

```
static void FetchImage( string url, string localFileName )
{
   using ( WebClient client = new WebClient() )
   {
      await client.DownloadFileTaskAsync( url, localFileName );
   }
}
```

TEKNOLOGISK
INSTITUT

Summary

- Introducing Task Parallel Library
- Combining and Handling Tasks
- Threading Issues

Question

TEKNOLOGISK INSTITUT

- You are creating an application with C# 5.0 containing a method named **`GetLine()`** with the following requirements:
  - The application must remain responsive with **`GetLine()`** executes
  - The **`GetLine()`** method set the **`_text`** member to the first line of the response
- Which code segments should be added?

```
01 public                                  void GetLine( WebResponse r )
02 {
03     var sr = new StreamReader( r.GetResponseStream() );
04     _text =                             sr.                          ;
05 }
```

`await`	`ReadLine()`	`ReadToEnd()`
`Task`	`async`	`ReadLineAsync()`
`ReadToEndAsync()`		`Task<string>`

TEKNOLOGISK
INSTITUT