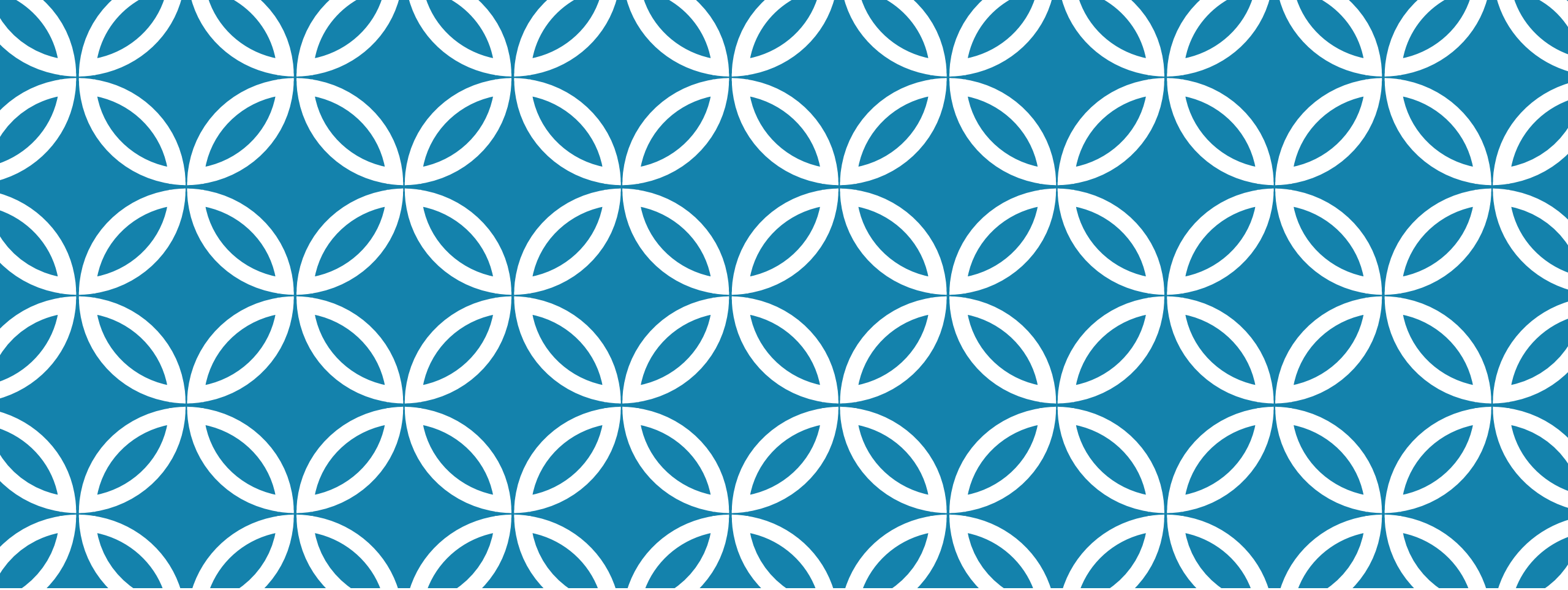
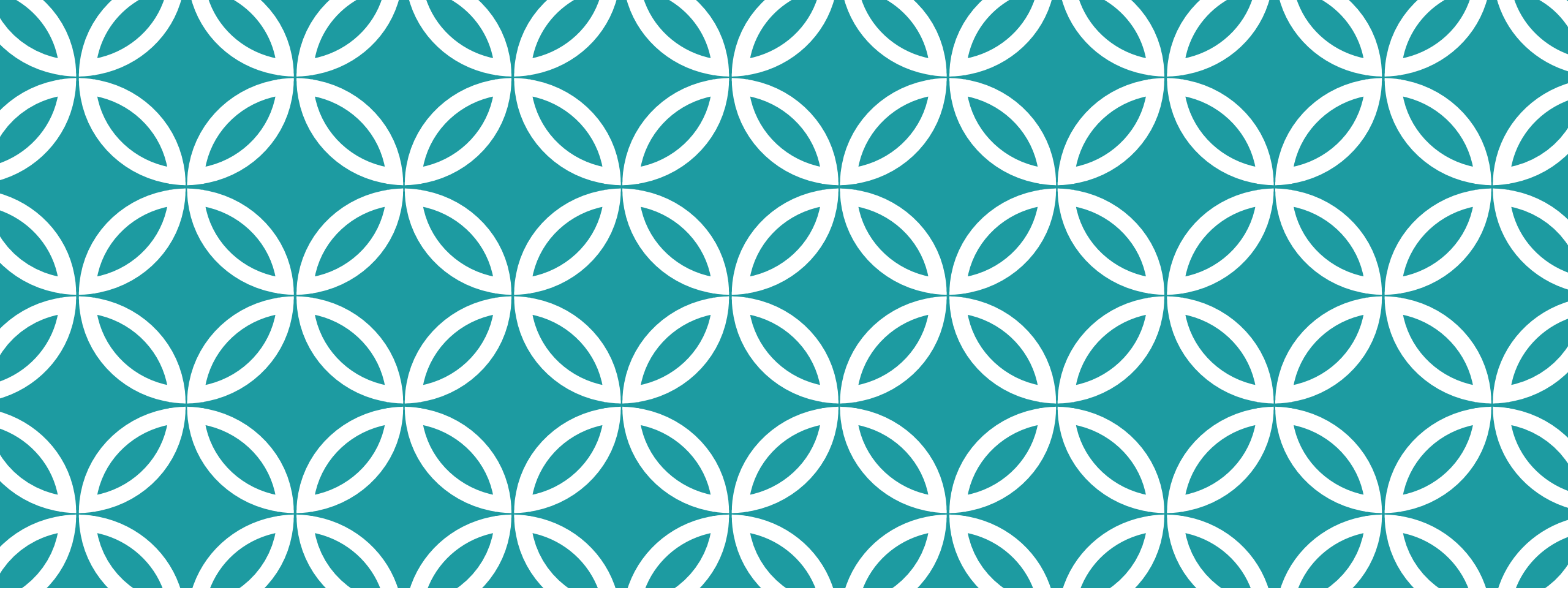


ATTRIBUTES
DYNAMIC PROGRAMMING
MULTITHREADING

Grundlæggende programmering
Lektion 11



ATTRIBUTES

Information om adfærd

ATTRIBUTES

En attribut er en deklarativ tag, der bruges til at formidle information til runtime om adfærd af forskellige elementer som klasser, metoder, strukturer, enumerators, samlinger etc. i et program.

Man kan tilføje deklarativ information til et program ved hjælp af en attribut.

Et deklarativt tag er vist ved firkantede ([]) parenteser placeret over elementet det anvendes til.

Attributter bruges til at tilføje metadata, såsom compiler instruktion og andre oplysninger, såsom kommentarer, beskrivelse, metoder og klasser til et program.

.NET Frameworket indeholder to typer af attributter: de foruddefinerede attributter (pre-defined) og specialbyggede attributter (custom built).

ATTRIBUTES

En attribut angives

```
[attribute(positional_parameters, name_parameter = value, ...)] element
```

Navn på den attribut, og dens værdier er angivet i de firkantede parenteser, før det element, som attributten anvendes på.

Positionelle parametre angiver de væsentlige oplysninger og navnet parametre angiver valgfri oplysninger.

.Net Frameworket tilbyder tre prædefinerede attributer:

- AttributeUsage
- Conditional
- Obsolete

ATTRIBUTES
ATTRIBUTEUSAGE

Den præ-definerede attribut AttributeUsage beskriver hvordan en brugerdefineret attribut klasse kan bruges.

- Den specificerer de typer af elementer, som attributten kan anvendes på.

Syntaks for at specifcere denne attribute

```
[AttributeUsage( validon, AllowMultiple=allowmultiple, Inherited=inherited )]
```

Parameteren validon specificerer de sprog elementerne som attribut kan placeres på. Det er en kombination af værdien af en enumerator *AttributeTargets*. Standard værdien er *AttributeTargets.All*.

Parameteren allowmultiple (valgfri) giver værdi til *AllowMultiple* property for denne attribut, en boolsk værdi. Hvis dette er sandt, er attributten Multiuse. Standard er falsk (engangsbrug).

ATTRIBUTES
ATTRIBUTEUSAGE

Parameteren inherited (valgfri) giver værdi til *Inherited* property for denne attribut, en Boolean værdi. Hvis det er sandt, bliver attributten arvet af afledte klasser. Standardværdien er false (ikke ned-arvet).

Et samlet eksempel:

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property, AllowMultiple = true)]
```

ATTRIBUTES

ATTRIBUTEUSAGE

CONDITIONAL

Denne prædefinerede attribut markerer en conditional method hvis udførelse afhænger af en angivet preprocessing identifier.

Det forårsager betinget kompilering af metodekald, afhængigt af den angivne værdi, såsom Debug eller Trace. For eksempel, viser værdierne af variablerne mens en kode debugges.

Syntaks

```
[Conditional( conditionalSymbol )]
```

For eksempel

```
[Conditional("DEBUG")]
```

ATTRIBUTES

ATTRIBUTEUSAGE

CONDITIONAL

```
using System;
using System.Diagnostics;

3 references
public class Myclass
{
    [Conditional("DEBUG")]
    3 references
    public static void Message(string msg)
    {
        Console.WriteLine(msg);
    }
}

0 references
class Test
{
    1 reference
    static void function1()
    {
        Myclass.Message("In Function 1.");
        function2();
    }
    1 reference
    static void function2()
    {
        Myclass.Message("In Function 2.");
    }

    0 references
    public static void Main()
    {
        Myclass.Message("In Main function.");
        function1();
        Console.ReadKey();
    }
}
```

ATTRIBUTES
ATTRIBUTEUSAGE OBSOLETE

Denne foruddefineret attribut markerer en program enhed, som ikke bør anvendes.

Det giver dig mulighed for at informere compileren om at kassere et bestemt mål element.

- For eksempel, når der anvendes en ny metode i en klasse, og hvis du stadig ønsker at beholde den gamle metode i klassen, kan du markere den som forældet ved at vise en meddelelse om at den nye metode bør anvendes i stedet for den gamle metode.

Syntaks

```
[Obsolete( message )]
[Obsolete( message, iserror )]
```

Hvor

- Parameteren *message* er en streng, der beskriver grunden til at elementet er forældet og hvad der skal bruges i stedet.
- Parameteren *iserror* er en boolesk værdi. Hvis dens værdi er sand, skal compileren behandle brugen af elementet som en fejl. Standardværdien er falsk (compiler genererer en advarsel).

ATTRIBUTES

ATTRIBUTEUSAGE

OBSOLETE

```
using System;

0 references
public class MyClass
{
    [Obsolete("Don't use OldMethod, use NewMethod instead", true)]
    1 reference
    static void OldMethod()
    {
        Console.WriteLine("It is the old method");
    }
    0 references
    static void NewMethod()
    {
        Console.WriteLine("It is the new method");
    }
    0 references
    public static void Main()
    {
        OldMethod();
    }
}
```

ATTRIBUTES
CUSTOM ATTRIBUTES

.NET frameworket tillader oprettelse af brugerdefinerede attributter, der kan bruges til at lagre deklarative oplysninger og kan hentes ved run-time. Denne information kan relateres til ethvert mål element afhængigt af designkriterier og program behov.

Oprettelse og brug af brugerdefinerede attributter involverer fire trin:

- Erklære en brugerdefineret attribut
- Konstruktion af den brugerdefinerede attribut
- Tilføj den brugerdefinerede attribut på et mål program element
- Adgang attributter gennem refleksion

Det sidste trin omfatter det at skrive et simpelt program til at læse igennem metadata til at finde forskellige notationer.
Metadata er data om data eller oplysninger, der anvendes til at beskrive andre data. Dette program bør bruge refleksioner til at få adgang attributter ved runtime.

ATTRIBUTES
CUSTOM ATTRIBUTES

En ny brugerdefineret attribute bør være afledt af System.Attribute klassen.

Herunder laver vi en brugerdefineret attribut kaldet DeBugInfo

```
//a custom attribute BugFix to be assigned to a class and
its members
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]


public class DeBugInfo : System.Attribute
```

ATTRIBUTES
CUSTOM ATTRIBUTES

Vi arbejder videre med DeBugInfo og lader den:

- Kode nummeret for bug
- Navnet på udvikleren der fandt bug
- Dato for det sidste kode review
- En streng besked til opbevaring af udviklerens bemærkninger

DeBugInfo klassen har tre private properties til at gemme de første tre informationer og en public property til at gemme beskeden.

- Derfor er bug number, udviklerens navn og dato for review lavet som positional parameters til DeBugInfo klassen og beskeden er en valgfri eller navngivet parameter.

Hver attribute skal have mindst én constructor.

- De positionelle parametre bør ledes gennem constructor.

Den udvidede *DeBugInfo* klasse kan ses på næste slide.

ATTRIBUTES
CUSTOM ATTRIBUTES

```
//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property, AllowMultiple = true)]

public class DeBugInfo : System.Attribute
{
        private int bugNo;
        private string developer;
        private string lastReview;
        public string message;
        public DeBugInfo(int bg, string dev, string d)
        {
                this.bugNo = bg;
                this.developer = dev;
                this.lastReview = d;
        }
        public int BugNo
        {
                get
                {
                        return bugNo;
                }
        }
```

ATTRIBUTES
CUSTOM ATTRIBUTES

```
        public string Developer
        {
                get
                {
                        return developer;
                }
        }

        public string LastReview
        {
                get
                {
                        return lastReview;
                }
        }

        public string Message
        {
                get
                {
                        return message;
                }
                set
                {
                        message = value;
                }
         }
}
```

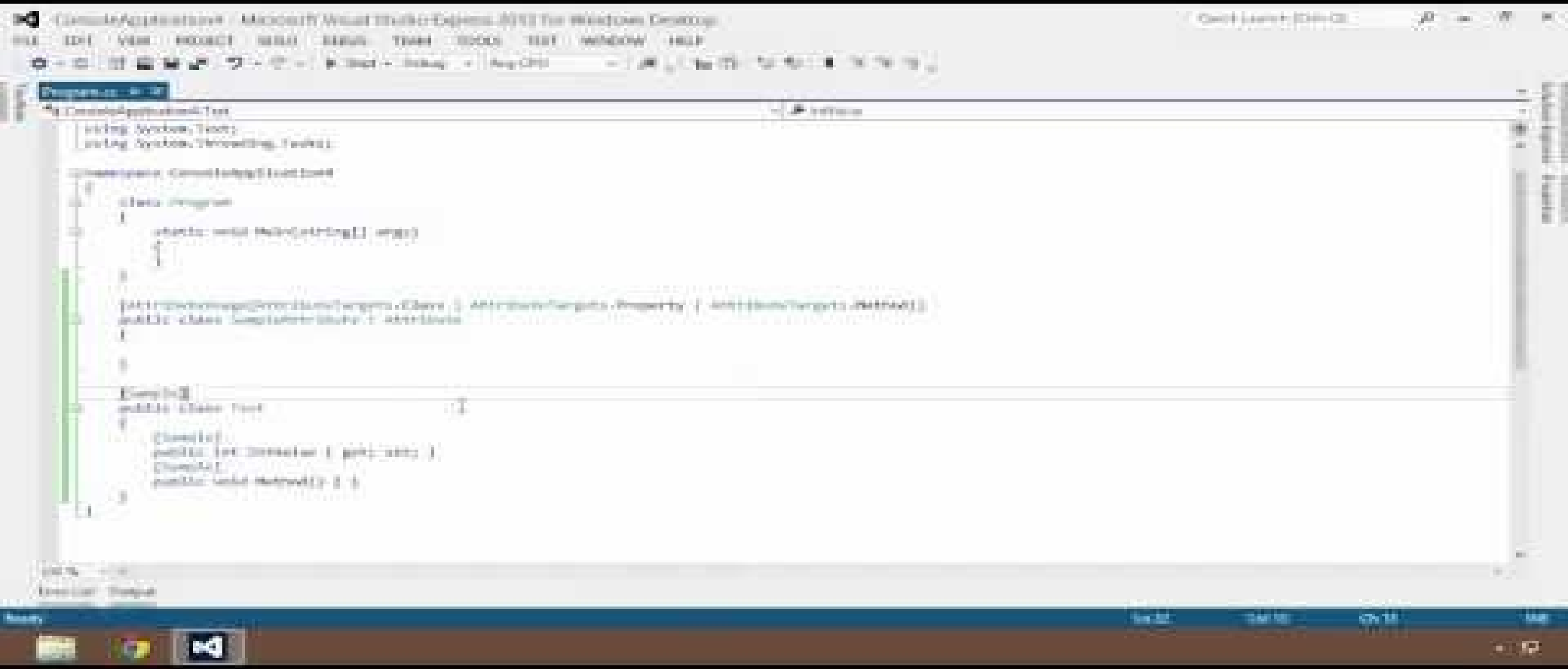
ATTRIBUTES
CUSTOM ATTRIBUTES

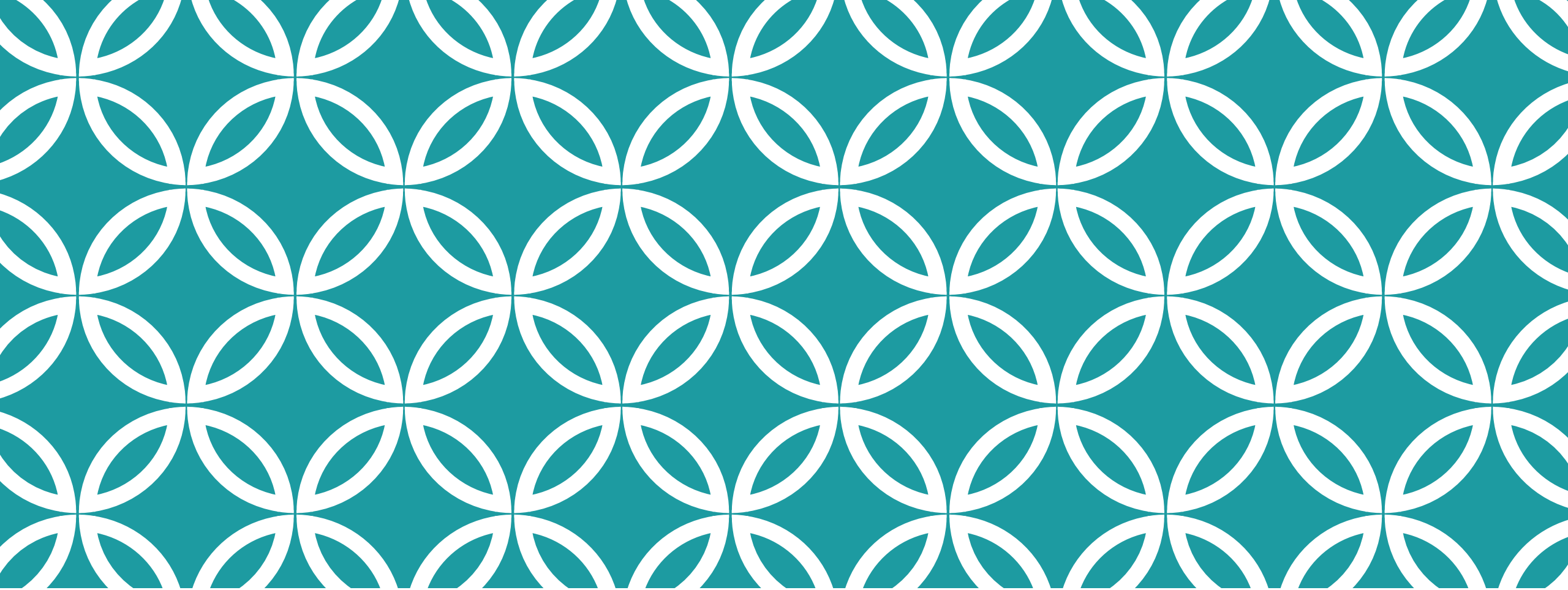
Attributten anvendes ved at placere den umiddelbart før sit mål:

```
[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
class Rectangle
{
        //member variable
        protected double length;
        protected double width;
        public Rectangle(double l, double w)
        {
                length = l;
                width = w;
        }
        [DeBugInfo(55, "Zara Ali", "19/10/2012", Message = "Return type mismatch")]

        public double GetArea()
        {
                return length * width;
        }
        [DeBugInfo(56, "Zara Ali", "19/10/2012")]

        public void Display()
        {
                Console.WriteLine("Length: {0}", length);
                Console.WriteLine("Width: {0}", width);
                Console.WriteLine("Area: {0}", GetArea());
        }
}
```





DYNAMIC PROGRAMMING

Reflektioner over indhold

DYNAMIC PROGRAMMING
REFLECTIONS

Reflection objekter anvendes til opnåelse af type information ved kørselstidspunktet. De klasser, der giver adgang til metadata i et kørende program er i System.Reflection namespace.

System.Reflection namespace indeholder klasser, der tillader en at få oplysninger om programmet og til dynamisk at tilføje typer, værdier og objekter til applikationen.

Reflection har de følgende brugsmønstre:

- Det giver view attribut information ved runtime.
- Det gør det muligt at undersøge forskellige typer i et assembly og at instantiere disse typer.
- Den tillader sen binding til metoder og egenskaber
- Det gør det muligt at skabe nye typer ved runtime og derefter udfører nogle opgaver ved hjælp af disse typer.

DYNAMIC PROGRAMMING

REFLECTIONS

VIEWING METADATA

MemberInfo objektet fra System.Reflection klassen skal initialiseres for at opdage attributter knyttet til en klasse.

- For at gøre dette definerer man et objekt af mål klassen, som:

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

```
1   using System;
2
3   [AttributeUsage(AttributeTargets.All)]
    2 references
4   public class HelpAttribute : System.Attribute
5   {
6       public readonly string Url;
7
        0 references
8       public string Topic   // Topic is a named parameter
9       {
10          get
11          {
12              return topic;
13          }
14          set
15          {
16              topic = value;
17          }
18      }
19
```

DYNAMIC PROGRAMMING

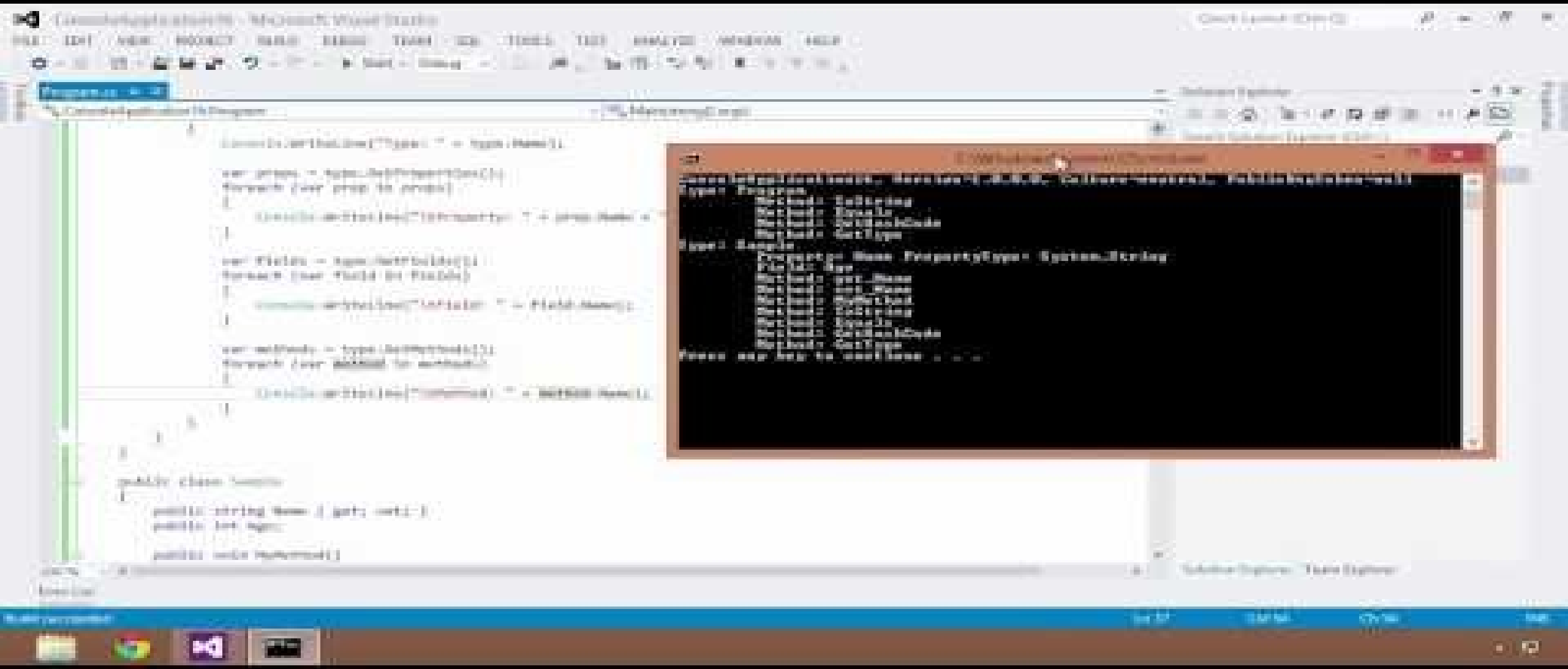
REFLECTIONS

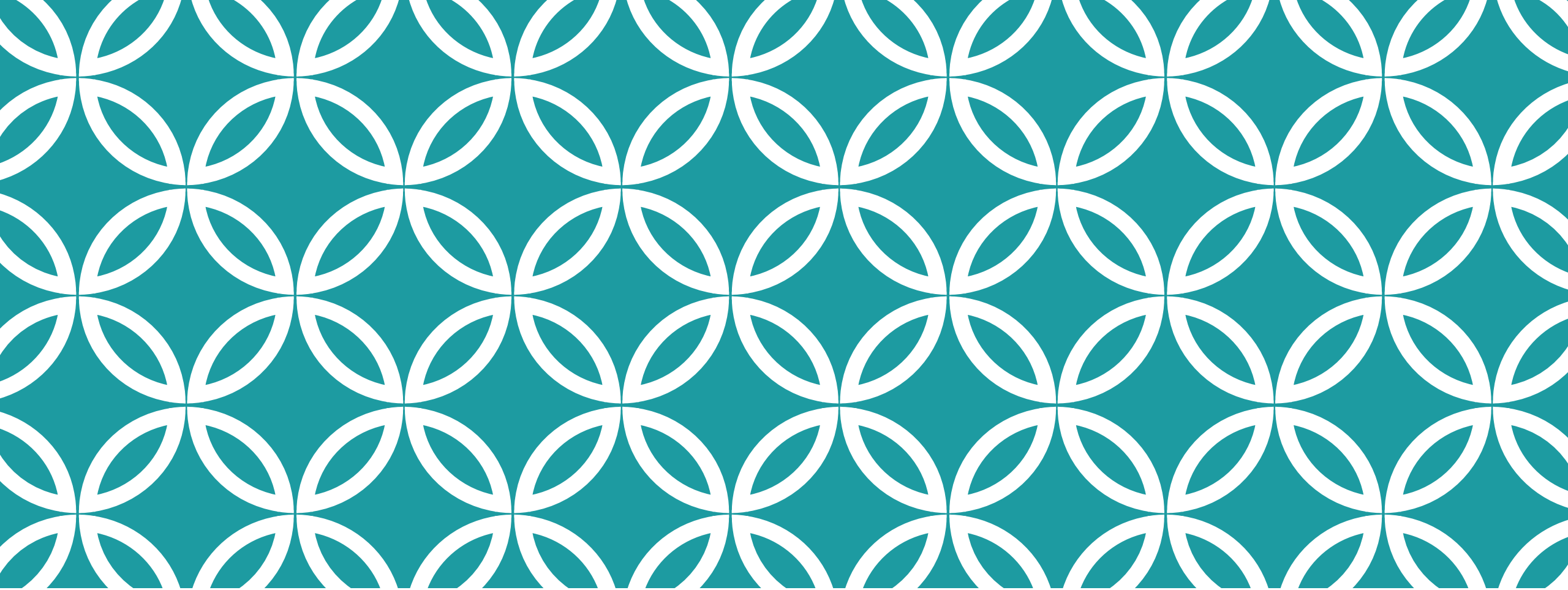
VIEWING METADATA

```
20      public HelpAttribute(string url)   // url is a positional parameter
21      {
22          this.Url = url;
23      }
24      private string topic;
25  }
26
27  [HelpAttribute("Information on the class MyClass")]
    1 reference
28  class MyClass
29  {
30  }
31  namespace AttributeAppl
32  {
        0 references
33      class Program
34      {
            0 references
35          static void Main(string[] args)
36          {
37              System.Reflection.MemberInfo info = typeof(MyClass);
38              object[] attributes = info.GetCustomAttributes(true);
39              for (int i = 0; i < attributes.Length; i++)
40              {
41                  System.Console.WriteLine(attributes[i]);
42              }
43
44              Console.ReadKey();
45          }
46      }
47  }
```

DYNAMIC PROGRAMMING
REFLECTIONS

Hent reflections.cs fra Fronter, den bygger videre på *DeBugInfo* eksemplet fra tidligere.


Type: Program
Method: ToString
Method: GetHashCode
Method: GetType
Type: Sample
Property: Name PropertyType: System.String
Field: Age
Method: get_Name
Method: set_Name
Method: SayHello
Method: ToString
Method: GetHashCode
Method: GetType
Press any key to continue . . .



MULTITHREADING

Flow af program udførelse

MULTITHREADING

En tråd er defineret som udførelsesvejen af et program.

Hver tråd definerer en unik strøm af kontrol.

- Hvis en applikation indebærer komplicerede og tidskrævende operationer, så er det ofte nyttigt at indstille forskellige udførelses stier eller tråde, hvor hver tråd udfører et bestemt job.

En tråd kaldes også en letvægts proces (lightweight process).

Et almindeligt eksempel på brug af tråde er gennemførelsen af parallel programmering af moderne operativsystemer.

Anvendelse af tråde sparer spild af CPU cyklus og øge effektiviteten af en applikation.

PROGRAMMERING AF TRÅDE

EN TRÅDS LIVSCYKLYS

En tråds livscyklus starter, når et objekt med System.Threading.Thread klassen skabes og slutter, når tråden er afsluttet eller fuldført.

Følgende er de forskellige stater i en tråds livscyklus:

- The Unstarted State: Det er den situation, hvor en forekomst af tråden er oprettet, men Start metoden er ikke kaldt.
- The Ready State: Det er den situation, hvor tråden er klar til at køre og venter på CPU cyklus.
- The Not Runnable State: En tråd er ikke eksekverbar når:
  - Sleep metoden er kaldt.
  - Wait metoden er kaldt.
  - Blokeret af I/O operationer.
- The Dead State: Det er den situation, hvor tråden fuldender udførelse eller afbrydes.

PROGRAMMERING AF TRÅDE

MAIN THREAD

I C # bliver System.Threading.Thread klassen brugt til at arbejde med tråde.

Det gør det muligt at skabe og adgang til de enkelte tråde i en flertrådede anmodninger.

Den første tråd der udføres i en proces kaldes den røde tråd.

Når et C # program starter udførelse bliver den røde tråd oprettet automatisk. Trådene oprettet ved hjælp af Thread klassen kaldes barne-tråde af den røde tråd.

Du kan få adgang en tråd ved hjælp af CurrentThread property af Thread klassen.

PROGRAMMERING AF TRÅDE

MAIN THREAD

```
1   using System;
2   using System.Threading;
3
4   namespace MultithreadingApplication
5   {
        0 references
6       class MainThreadProgram
7       {
            0 references
8           static void Main(string[] args)
9           {
10              Thread th = Thread.CurrentThread;
11              th.Name = "MainThread";
12              Console.WriteLine("This is {0}", th.Name);
13              Console.ReadKey();
14          }
15      }
16  }
```

PROGRAMMERING AF TRÅDE

MAIN THREAD

Property	Description
CurrentContext	Gets the current context in which the thread is executing.
CurrentCulture	Gets or sets the culture for the current thread.
CurrentPrinciple	Gets or sets the thread's current principal (for role-based security).
CurrentThread	Gets the currently running thread.
CurrentUICulture	Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run-time.
ExecutionContext	Gets an ExecutionContext object that contains information about the various contexts of the current thread.
IsAlive	Gets a value indicating the execution status of the current thread.

PROGRAMMERING AF TRÅDE

MAIN THREAD

IsBackground	Gets or sets a value indicating whether or not a thread is a background thread.
IsThreadPoolThread	Gets a value indicating whether or not a thread belongs to the managed thread pool.
ManagedThreadId	Gets a unique identifier for the current managed thread.
Name	Gets or sets the name of the thread.
Priority	Gets or sets a value indicating the scheduling priority of a thread.
ThreadState	Gets a value containing the states of the current thread.

PROGRAMMERING AF TRÅDE

Tråde laves ved at udvide Thread klassen. Den udvidede Thread klasse kalder derefter Start() metoden til at begynde barne-trådens udførelse.

```
1   using System;
2   using System.Threading;
3
4   namespace MultithreadingApplication
5   {
        0 references
6       class ThreadCreationProgram
7       {
            1 reference
8           public static void CallToChildThread()
9           {
10              Console.WriteLine("Child thread starts");
11          }
12
            0 references
13          static void Main(string[] args)
14          {
15              ThreadStart childref = new ThreadStart(CallToChildThread);
16              Console.WriteLine("In Main: Creating the Child thread");
17              Thread childThread = new Thread(childref);
18              childThread.Start();
19              Console.ReadKey();
20          }
21      }
22  }
```

PROGRAMMERING AF TRÅDE

ADMINISTRATION AF TRÅDE

Thread klassen giver forskellige metoder til håndtering af tråde. Det følgende eksempel demonstrerer anvendelsen af sleep() fremgangsmåden til fremstilling af en tråd pause i en bestemt tidsperiode.

```
1   using System;
2   using System.Threading;
3
4   namespace MultithreadingApplication
5   {
        0 references
6       class ThreadCreationProgram
7       {
            1 reference
8           public static void CallToChildThread()
9           {
10              Console.WriteLine("Child thread starts");
11
12              // the thread is paused for 5000 milliseconds
13              int sleepfor = 5000;
14
15              Console.WriteLine("Child Thread Paused for {0} seconds", sleepfor / 1000);
16              Thread.Sleep(sleepfor);
17              Console.WriteLine("Child thread resumes");
18          }
19
            0 references
20          static void Main(string[] args)
21          {
22              ThreadStart childref = new ThreadStart(CallToChildThread);
23              Console.WriteLine("In Main: Creating the Child thread");
24              Thread childThread = new Thread(childref);
25              childThread.Start();
26              Console.ReadKey();
27          }
28      }
29  }
```

PROGRAMMERING AF TRÅDE
ØDELÆGGELSE AF TRÅDE

Abort() metoden bruges til at ødelægge tråde.

Runtime afbryder tråden ved at kaste en ThreadAbortException. Denne undtagelse kan ikke blive fanget, styringen sendes til en *finally* blok, hvis nogen.

```
1   using System;
2   using System.Threading;
3
4   namespace MultithreadingApplication
5   {
        0 references
6       class ThreadCreationProgram
7       {
            1 reference
8           public static void CallToChildThread()
9           {
10              try
11              {
12                  Console.WriteLine("Child thread starts");
13
14                  // do some work, like counting to 10
15                  for (int counter = 0; counter <= 10; counter++)
16                  {
17                      Thread.Sleep(500);
18                      Console.WriteLine(counter);
19                  }
20
21                  Console.WriteLine("Child Thread Completed");
22              }
```

PROGRAMMERING AF TRÅDE
ØDELÆGGELSE AF TRÅDE

Abort() metoden bruges til at ødelægge tråde.

Runtime afbryder tråden ved at kaste en ThreadAbortException. Denne undtagelse kan ikke blive fanget, styringen sendes til en *finally* blok, hvis nogen.

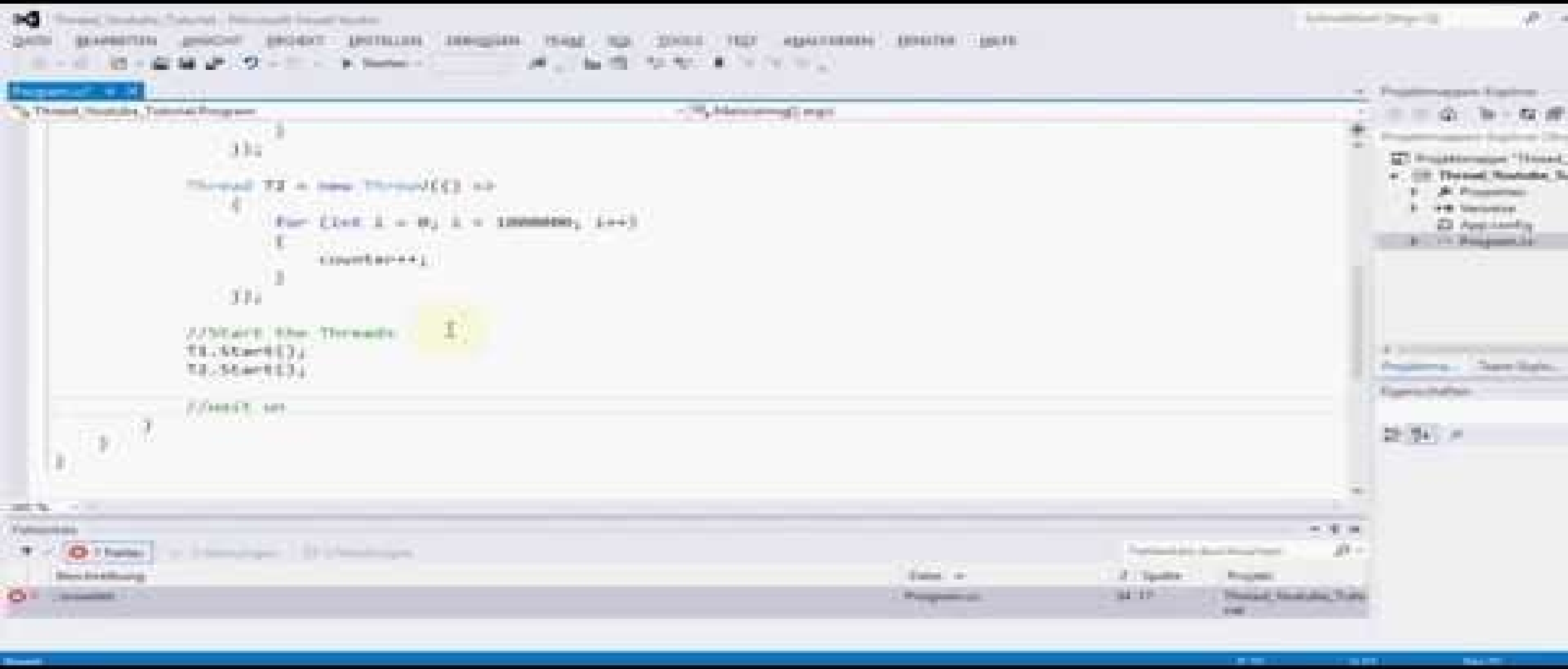
```
23
24              catch (ThreadAbortException e)
25              {
26                  Console.WriteLine("Thread Abort Exception");
27              }
28              finally
29              {
30                  Console.WriteLine("Couldn't catch the Thread Exception");
31              }
32          }
33
            0 references
34          static void Main(string[] args)
35          {
36              ThreadStart childref = new ThreadStart(CallToChildThread);
37              Console.WriteLine("In Main: Creating the Child thread");
38              Thread childThread = new Thread(childref);
39              childThread.Start();
40
41              //stop the main thread for some time
42              Thread.Sleep(2000);
43
44              //now abort the child
45              Console.WriteLine("In Main: Aborting the Child thread");
46
47              childThread.Abort();
48              Console.ReadKey();
49          }
50      }
51  }
```

PROGRAMMERING AF TRÅDE
MUTUAL EXCLUSION

I datalogi er gensidig udelukkelse (mutual exclusion) en egenskab ved concurrency kontrol, som er lavet med henblik på at forebygge race condition / hazard.

- Race condition er når output er afhængig af sekvensen eller timingen af andre ukontrollerbare begivenheder.

Mutual exclusion er kravet om, at en udførelses-tråd aldrig når sin kritiske strækning på samme tid som en anden samtidig tråd udførelse går ind i sin egen kritiske sektion.



LEKTIE

Kig på dette til næste gang

LEKTIE

Læs:

- https://www.tutorialspoint.com/csharp/csharp_properties.htm
- https://www.tutorialspoint.com/csharp/csharp_multithreading.htm

Opgave

- Gå i gang med jeres eksamensopgave

KILDER

Materiale benyttet i denne lektion
Noget af det er udover pensum-listen!

KILDER

Attributes

- https://www.tutorialspoint.com/csharp/csharp_attributes.htm
- https://youtu.be/mGkO7oM5VI8

Dynamic programming

- https://www.tutorialspoint.com/csharp/csharp_reflection.htm
- https://msdn.microsoft.com/en-us/library/hh156524(v=vs.110).aspx
- https://youtu.be/3FvT6uNMT7M (Reflection)

Multithreading

- https://www.tutorialspoint.com/csharp/csharp_multithreading.htm
- https://msdn.microsoft.com/en-us/library/aa645740(v=vs.71).aspx
- https://youtu.be/bxJzqNCZsNw (Mutex)