Threads, processes, parallel programming,

AppDomains and async call in .NET 4.5 /5.0

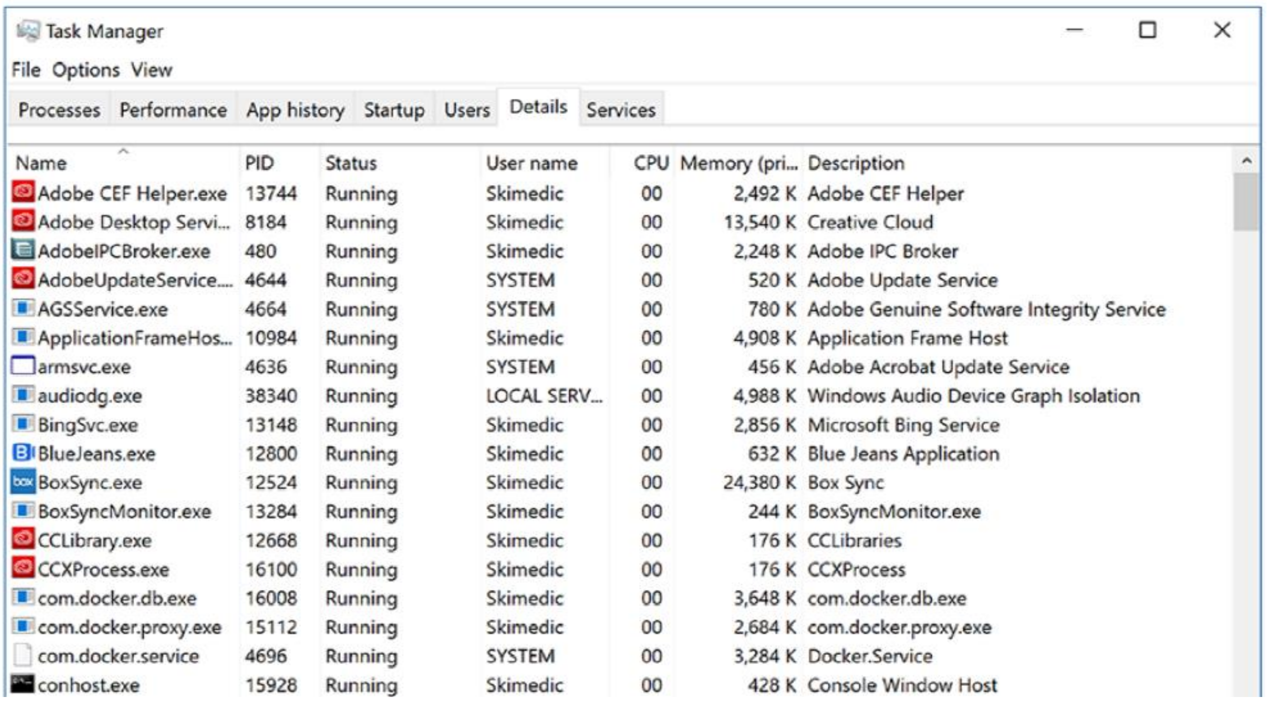
NuGet packages

Processes and AppDomains

- Process defined:
    - an ”active” application
    - A set of resources and memory allocation
    - Runs in a ”sandbox” by the operating system (OS)

General about processes and threads (Win32)

- OS creates a separate process (loaded into memory)
    - gives stability on run-time
    - and can (normally) not have an impact on other applications or OS
- Every process has a PID
- Every process has at least one thread (Main)


Task Manager
File Options View
Processes Performance App history Startup Users Details Services

Name	PID	Status	User name	CPU	Memory (pri...	Description
Adobe CEF Helper.exe	13744	Running	Skimedic	00	2,492 K	Adobe CEF Helper
Adobe Desktop Servi...	8184	Running	Skimedic	00	13,540 K	Creative Cloud
AdobeIPCBroker.exe	480	Running	Skimedic	00	2,248 K	Adobe IPC Broker
AdobeUpdateService....	4644	Running	SYSTEM	00	520 K	Adobe Update Service
AGSService.exe	4664	Running	SYSTEM	00	780 K	Adobe Genuine Software Integrity Service
ApplicationFrameHos...	10984	Running	Skimedic	00	4,908 K	Application Frame Host
armsvc.exe	4636	Running	SYSTEM	00	456 K	Adobe Acrobat Update Service
audiodg.exe	38340	Running	LOCAL SERV...	00	4,988 K	Windows Audio Device Graph Isolation
BingSvc.exe	13148	Running	Skimedic	00	2,856 K	Microsoft Bing Service
BlueJeans.exe	12800	Running	Skimedic	00	632 K	Blue Jeans Application
BoxSync.exe	12524	Running	Skimedic	00	24,380 K	Box Sync
BoxSyncMonitor.exe	13284	Running	Skimedic	00	244 K	BoxSyncMonitor.exe
CCLibrary.exe	12668	Running	Skimedic	00	176 K	CCLibraries
CCXProcess.exe	16100	Running	Skimedic	00	176 K	CCXProcess
com.docker.db.exe	16008	Running	Skimedic	00	3,648 K	com.docker.db.exe
com.docker.proxy.exe	15112	Running	Skimedic	00	2,684 K	com.docker.proxy.exe
com.docker.service	4696	Running	SYSTEM	00	3,284 K	Docker.Service
conhost.exe	15928	Running	Skimedic	00	428 K	Console Window Host

The roles of threads in applications

- If a process has a single thread
    - A wrong while construction can for instance lock the application
        - GUI can not be contacted
        - Is on the other page ”thread safe”: data is not shared between more threads
    - It is possible to create worker threads
    - Every thread has its own path
    - Pseudo – parallism

Limits for threads

- The CPU makes a context switch between active threads
- This switch takes time from the active work in a thread
- OS uses often the ”time-slice” principle (All machines with single CPU)
- Every thread process is offered a quantum
- The thread process stores data and PC (Program Counter) when stopped
    - Is called TLS (Thread Local Storage)

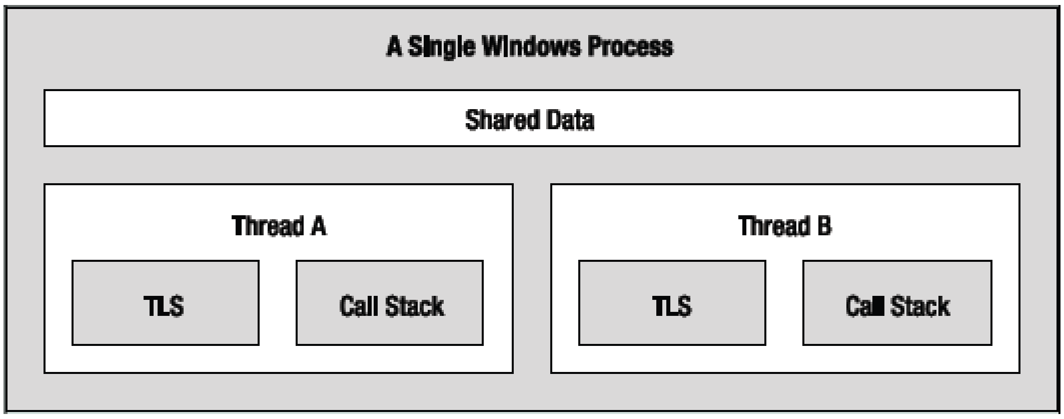


Figure 17-2. The Windows process/thread relationship

Processes under the .NET platform

- **System.Diagnostics namespace**
    - creates processes
    - Diagnostics

Process-Centric Types of the System.Diagnostics Namespace	Meaning in Life
Process	The Process class provides access to local and remote processes and also allows you to programmatically start and stop processes.
ProcessModule	This type represents a module (*.dll or *.exe) that is loaded into a particular process. Understand that the ProcessModule type can represent any module—COM-based, .NET-based, or traditional C-based binaries.
ProcessModuleCollection	This provides a strongly typed collection of ProcessModule objects.
ProcessStartInfo	This specifies a set of values used when starting a process via the Process.Start() method.
ProcessThread	This type represents a thread within a given process. Be aware that ProcessThread is a type used to diagnose a process's thread set and is not used to spawn new threads of execution within a process.
ProcessThreadCollection	This provides a strongly typed collection of ProcessThread objects.

- The Process Type
  - Properties

***Table 17-2**. Select Properties of the Process Type*

Property	Meaning in Life
ExitTime	This property gets the timestamp associated with the process that has terminated (represented with a DateTime type).
Handle	This property returns the handle (represented by an IntPtr) associated to the process by the OS. This can be useful when building .NET applications that need to communicate with unmanaged code.
Id	This property gets the PID for the associated process.
MachineName	This property gets the name of the computer the associated process is running on.
MainWindowTitle	MainWindowTitle gets the caption of the main window of the process (if the process does not have a main window, you receive an empty string).

Modules	This property provides access to the strongly typed ProcessModuleCollection type, which represents the set of modules (*.dll or *.exe) loaded within the current process.
ProcessName	This property gets the name of the process (which, as you would assume, is the name of the application itself).
Responding	This property gets a value indicating whether the user interface of the process is responding to user input (or is currently “hung”).
StartTime	This property gets the time that the associated process was started (via a DateTime type).
Threads	This property gets the set of threads that are running in the associated process (represented via a collection of ProcessThread objects).

- Methods

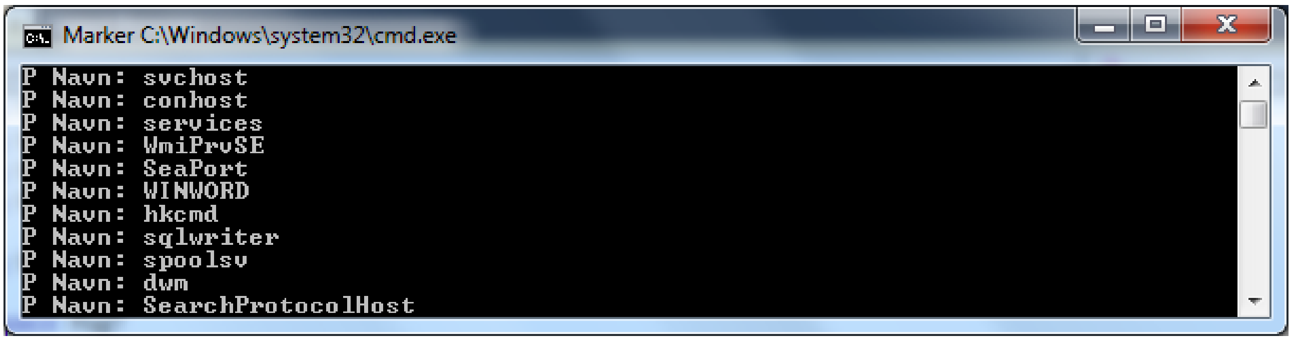
Table 17-3*. Select Methods of the Process Type*

Method	Meaning in Life
CloseMainWindow()	This method closes a process that has a user interface by sending a close message to its main window.
GetCurrentProcess()	This static method returns a new Process object that represents the currently active process.
GetProcesses()	This static method returns an array of new Process objects running on a given machine.
Kill()	This method immediately stops the associated process.
Start()	This method starts a process.

Example:

```
using System;
using System.Diagnostics;


public class VisProcess
{


  public static void Main()

  {

        Process[] pro = Process.GetProcesses();
        foreach ( Process p in pro)
        {
          string proc = string.Format("P Navn: {0}",
                                     p.ProcessName);
            Console.WriteLine(proc);

        }
        Console.ReadLine();

  }




}
```

Output (partly):

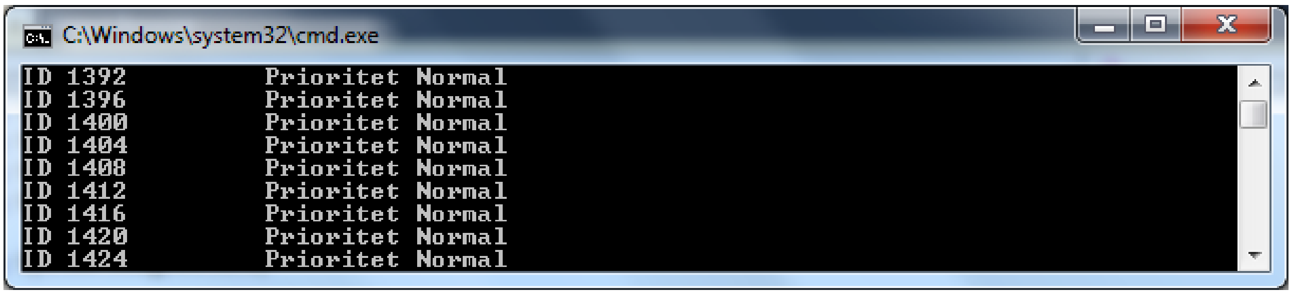


Marker C:\Windows\system32\cmd.exe

```
P Navn: svchost
P Navn: conhost
P Navn: services
P Navn: WmiPrvSE
P Navn: SeaPort
P Navn: WINWORD
P Navn: hkcmd
P Navn: sqlwriter
P Navn: spoolsv
P Navn: dwm
P Navn: SearchProtocolHost
```

Which process owns which threads ?

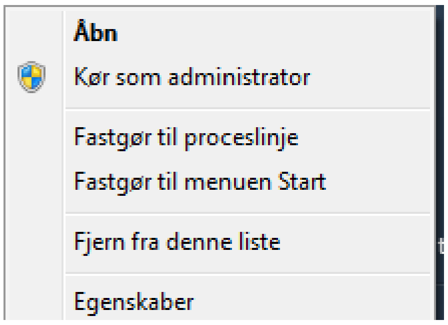
```
using System;
using System.Diagnostics;


public class VisProcess
{


  public static void Main()

  {

        Process[] pro = Process.GetProcesses();
        foreach ( Process p in pro)
        {
            findTråde(p.Id);


        }
        Console.ReadLine();

  }

   public static void findTråde(int pId) {

          Process proc= null;

          try {

            proc = Process.GetProcessById(pId);

          }catch(Exception e)
          {
             Console.WriteLine(e.Message);
          }


          ProcessThreadCollection tr = proc.Threads;
          foreach(ProcessThread pt in tr)
          {
             string inf = string.Format("ID {0}
                          \tPrioritet {1}",
                          pt.Id,pt.PriorityLevel);


             Console.WriteLine(inf);
           }
    }

}
```



```
ID 1392         Prioritet Normal
ID 1396         Prioritet Normal
ID 1400         Prioritet Normal
ID 1404         Prioritet Normal
ID 1408         Prioritet Normal
ID 1412         Prioritet Normal
ID 1416         Prioritet Normal
ID 1420         Prioritet Normal
ID 1424         Prioritet Normal
```

- Gives info about threads

- Notice: ”Run as Administrator”



ProcessThread type

Table 17-4. Select Members of the ProcessThread Type

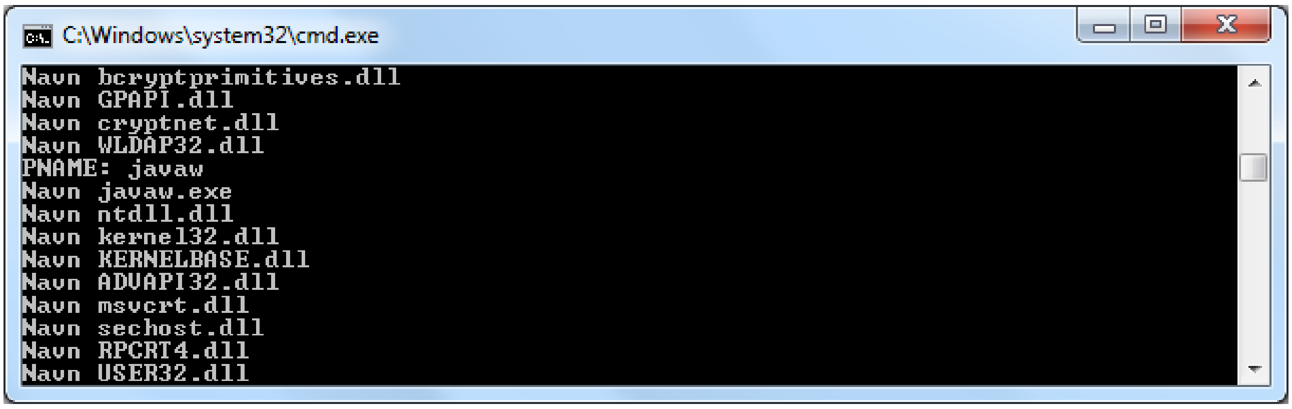
Member	Meaning in Life
CurrentPriority	Gets the current priority of the thread
Id	Gets the unique identifier of the thread
IdealProcessor	Sets the preferred processor for this thread to run on
PriorityLevel	Gets or sets the priority level of the thread
ProcessorAffinity	Sets the processors on which the associated thread can run
StartAddress	Gets the memory address of the function that the operating system called that started this thread
StartTime	Gets the time that the operating system started the thread
ThreadState	Gets the current state of this thread
TotalProcessorTime	Gets the total amount of time that this thread has spent using the processor
WaitReason	Gets the reason that the thread is waiting

Process modules – Which DLL's are used by a process?

```
using System;
using System.Diagnostics;


public class VisProcess
{


  public static void Main()

  {

        Process[] pro = Process.GetProcesses();

        foreach ( Process p in pro)
        {
            findTråde(p.Id);


        }
        Console.ReadLine();

  }


   public static void findTråde(int pId) {

          Process proc= null;

          try {

            proc = Process.GetProcessById(pId);

          }catch(Exception e)
          {
             Console.WriteLine(e.Message);
          }
          Console.WriteLine("PNAME: {0}",
                      proc.ProcessName);
          ProcessModuleCollection pmc = proc.Modules;
          foreach(ProcessModule pm in pmc)
          {
             string inf = string.Format("Navn {0} ",
                                          pm.ModuleName);

             Console.WriteLine(inf);
```

```
                }
        }


}
```

Output (partly):



```
Navn bcryptprimitives.dll
Navn GPAPI.dll
Navn cryptnet.dll
Navn WLDAP32.dll
PNAME: javaw
Navn javaw.exe
Navn ntdll.dll
Navn kernel32.dll
Navn KERNELBASE.dll
Navn ADVAPI32.dll
Navn msvcrt.dll
Navn sechost.dll
Navn RPCRT4.dll
Navn USER32.dll
```

To start and stop a process

Example:

```
...
public  void StartVerdensprogrammet()
  {

            Process vp = Process.Start("notepad.exe");
            Console.ReadLine();
            try
            {
              vp.Kill();
            }catch(Exception e)
            {
             Console.WriteLine(e.Message);
            }


  }
...
```

- ProcessStartInfo

```
public sealed class ProcessStartInfo : object
{
  public ProcessStartInfo();
  public ProcessStartInfo(string fileName);
  public ProcessStartInfo(string fileName, string arguments);
  public string Arguments { get; set; }
  public bool CreateNoWindow { get; set; }
  public StringDictionary EnvironmentVariables { get; }
  public bool ErrorDialog { get; set; }
  public IntPtr ErrorDialogParentHandle { get; set; }
  public string FileName { get; set; }
  public bool LoadUserProfile { get; set; }
  public SecureString Password { get; set; }
  public bool RedirectStandardError { get; set; }
  public bool RedirectStandardInput { get; set; }
  public bool RedirectStandardOutput { get; set; }
  public Encoding StandardErrorEncoding { get; set; }
  public Encoding StandardOutputEncoding { get; set; }
  public bool UseShellExecute { get; set; }
  public string Verb { get; set; }
  public string[] Verbs { get; }
  public ProcessWindowStyle WindowStyle { get; set; }
  public string WorkingDirectory { get; set; }
}
```

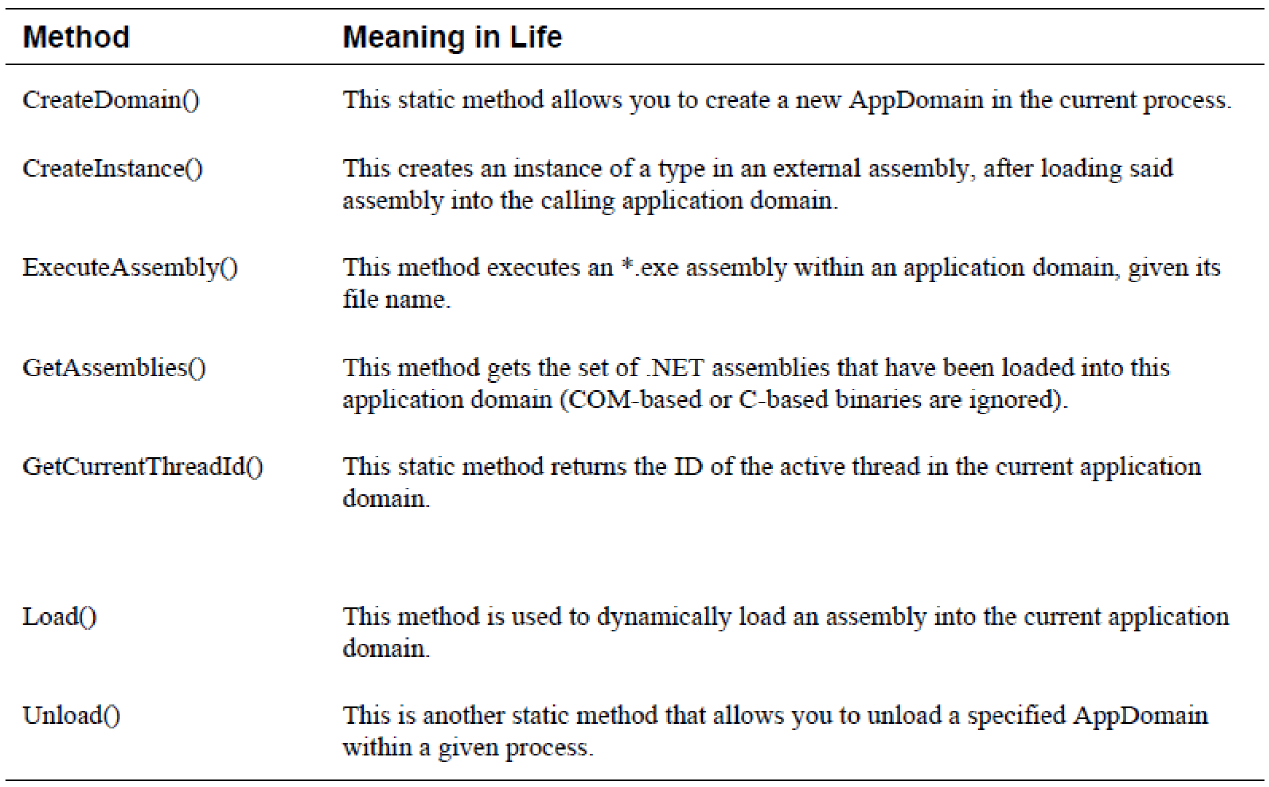
.NET Application domain

- .NET assemblies are logical partitions in a process
    - This is understood as an **AppDomain**

- Advantage:
- AppDomains are OS neutral
- AppDomains are ”less expensive”.
    - CLR can load / unload AppDomains
    - If an AppDomain crash the rest can function normal

- Using programming a Dll can be **loaded / unloaded**
    - Methods:

Table 17-5. *Select Methods of AppDomain*



Method	Meaning in Life
CreateDomain()	This static method allows you to create a new AppDomain in the current process.
CreateInstance()	This creates an instance of a type in an external assembly, after loading said assembly into the calling application domain.
ExecuteAssembly()	This method executes an *.exe assembly within an application domain, given its file name.
GetAssemblies()	This method gets the set of .NET assemblies that have been loaded into this application domain (COM-based or C-based binaries are ignored).
GetCurrentThreadId()	This static method returns the ID of the active thread in the current application domain.
Load()	This method is used to dynamically load an assembly into the current application domain.
Unload()	This is another static method that allows you to unload a specified AppDomain within a given process.

- Properties:

Table 17-6. *Select Properties of AppDomain*

Property	Meaning in Life
BaseDirectory	This gets the directory path that the assembly resolver uses to probe for assemblies.
CurrentDomain	This static property gets the application domain for the currently executing thread.
FriendlyName	This gets the friendly name of the current application domain.
MonitoringIsEnabled	This gets or sets a value that indicates whether CPU and memory monitoring of application domains is enabled for the current process. Once monitoring is enabled for a process, it cannot be disabled.
SetupInformation	This gets the configuration details for a given application domain, represented by an AppDomainSetup object.

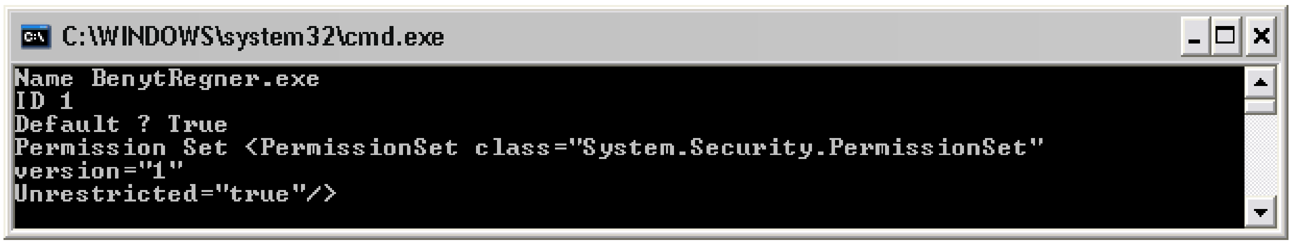
- Events:

Table 17-7. *Select Events of the AppDomain Type*

Event	Meaning in Life
AssemblyLoad	This occurs when an assembly is loaded into memory.
AssemblyResolve	This event will fire when the assembly resolver cannot find the location of a required assembly.
DomainUnload	This occurs when an AppDomain is about to be unloaded from the hosting process.
FirstChanceException	This event allows you to be notified that an exception has been thrown from the application domain, before the CLR will begin looking for a fitting catch statement.
ProcessExit	This occurs on the default application domain when the default application domain's parent process exits.
UnhandledException	This occurs when an exception is not caught by an exception handler.

Info about AppDomains using C# code

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;



namespace BenytRegner
{
    class Program
    {
        static void Main(string[] args)
        {
            AppDomain theDefault = AppDomain.CurrentDomain;
            Console.WriteLine("Name {0}", theDefault.FriendlyName);
            Console.WriteLine("ID {0}", theDefault.Id);
            Console.WriteLine("Default ? {0}", theDefault.IsDefaultAppDomain());
            Console.WriteLine("Permission Set {0}", theDefault.PermissionSet);

        }
    }
}
```



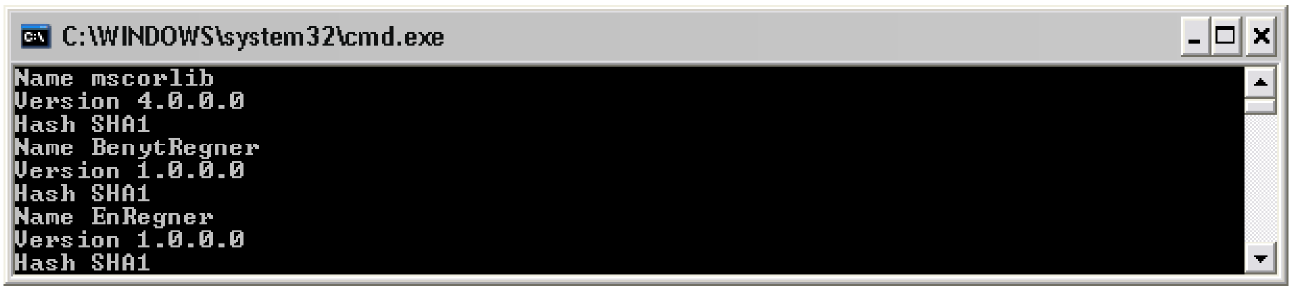
```
Name BenytRegner.exe
ID 1
Default ? True
Permission Set <PermissionSet class="System.Security.PermissionSet"
version="1"
Unrestricted="true"/>
```

Enumerating loaded assemblies

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;



namespace BenytRegner
{
    class Program
    {
        static void Main(string[] args)
        {
            AppDomain theDefault = AppDomain.CurrentDomain;
```

Remember name-space

```
            Assembly[] listOfAsm = theDefault.GetAssemblies();

            foreach (Assembly a in listOfAsm)
            {
                Console.WriteLine("Name {0}", a.GetName().Name);
                Console.WriteLine("Version {0}", a.GetName().Version);
                Console.WriteLine("Hash {0}", a.GetName().HashAlgorithm);

            }

        }


    }
}
```


C:\WINDOWS\system32\cmd.exe

```
Name mscorlib
Version 4.0.0.0
Hash SHA1
Name BenytRegner
Version 1.0.0.0
Hash SHA1
Name EnRegner
Version 1.0.0.0
Hash SHA1
```

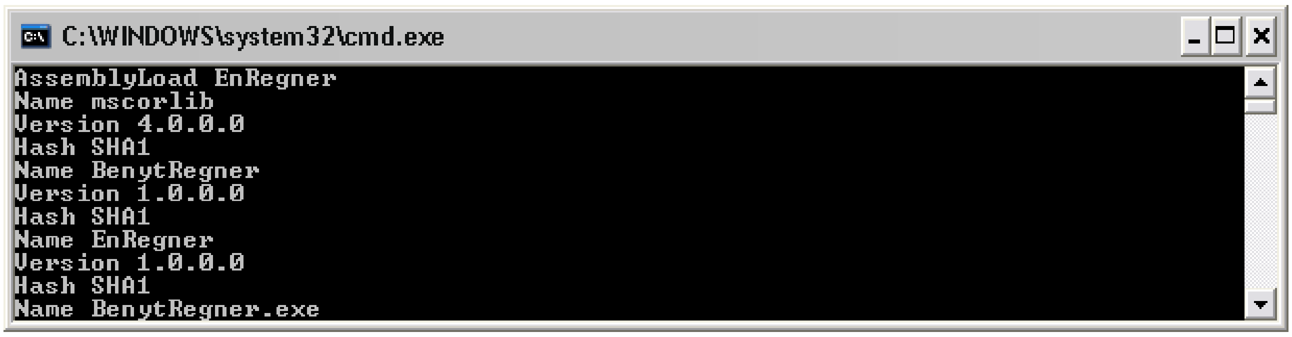
Notification with an event when an assembly is loaded

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;



namespace BenytRegner
{
    class Program
    {
        static void Main(string[] args)
        {
            AppDomain theDefault = AppDomain.CurrentDomain;

            theDefault.AssemblyLoad += (o, s) =>
                {
                    Console.WriteLine("AssemblyLoad {0}", s.LoadedAssembly.GetName().Name);
                };

            Assembly asm = Assembly.Load("EnRegner");

            Assembly[] listOfAsm = theDefault.GetAssemblies();
```

Event

Load

```
            foreach (Assembly a in listOfAsm)
            {
                Console.WriteLine("Name {0}", a.GetName().Name);
                Console.WriteLine("Version {0}", a.GetName().Version);
                Console.WriteLine("Hash {0}", a.GetName().HashAlgorithm);

            }

            Console.WriteLine("Name {0}", theDefault.FriendlyName);
            Console.WriteLine("ID {0}", theDefault.Id);
            Console.WriteLine("Default ? {0}", theDefault.IsDefaultAppDomain());
            Console.WriteLine("Permission Set {0}", theDefault.PermissionSet);

        }


    }
}
```



```
AssemblyLoad EnRegner
Name mscorlib
Version 4.0.0.0
Hash SHA1
Name BenytRegner
Version 1.0.0.0
Hash SHA1
Name EnRegner
Version 1.0.0.0
Hash SHA1
Name BenytRegner.exe
```

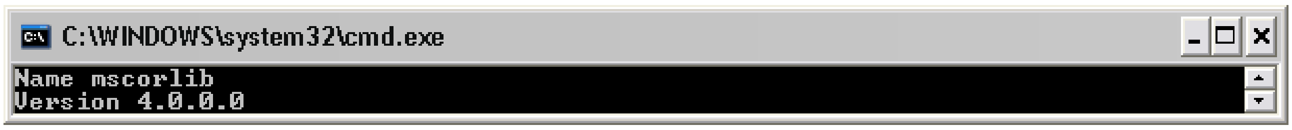
- The event is called when the assembly is loaded

To create new AppDomains

- (DLL not loaded yet)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;



namespace BenytRegner
{
    class Program
    {
     static void Main(string[] args)
     {
```

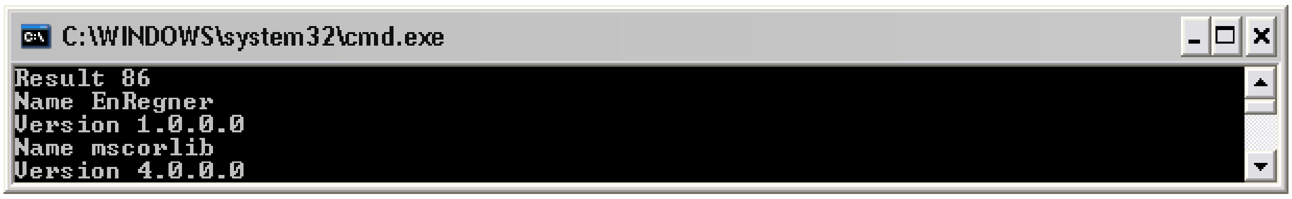
```
     AppDomain aNewDomain = AppDomain.CreateDomain("aNewDomain");


      var loadAsms = from a in aNewDomain.GetAssemblies() orderby a.GetName().Name select a;

            foreach (var a in loadAsms)
            {
                Console.WriteLine("Name {0}", a.GetName().Name);
                Console.WriteLine("Version {0}", a.GetName().Version);

            }

        }

    }
}
```



```
Name mscorlib
Version 4.0.0.0
```

Load assembly dynamically in an AppDomain

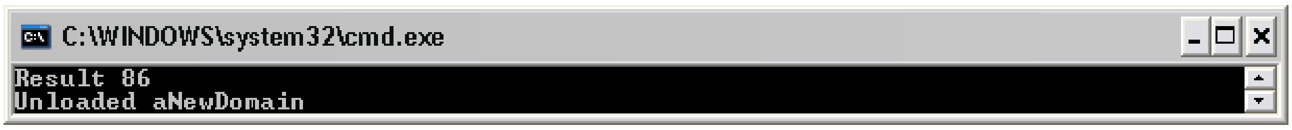
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;



namespace BenytRegner
{
    class Program
    {
        static void Main(string[] args)
        {


            AppDomain aNewDomain = AppDomain.CreateDomain("aNewDomain");
            Assembly asm = aNewDomain.Load("EnRegner");
            Type type = asm.GetType("EnRegner.EnRegner");
            object regner = asm.CreateInstance("EnRegner.EnRegner");

            MethodInfo mi = type.GetMethod("Plus");
            object[] argsToMethod = {40.4,45.6};
            object ores = mi.Invoke(regner, argsToMethod);

            Console.WriteLine("Result {0}", (double)ores);
            var loadAsms =
            from a in aNewDomain.GetAssemblies() orderby a.GetName().Name select a;

            foreach (var a in loadAsms)
            {
                Console.WriteLine("Name {0}", a.GetName().Name);
                Console.WriteLine("Version {0}", a.GetName().Version);


            }



        }


    }
}
```


C:\WINDOWS\system32\cmd.exe

```
Result 86
Name EnRegner
Version 1.0.0.0
Name mscorlib
Version 4.0.0.0
```

Unload an assembly dynamically in an AppDomain

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;



namespace BenytRegner
{
    class Program
    {
        static void Main(string[] args)
        {
            AppDomain aNewDomain = AppDomain.CreateDomain("aNewDomain");
            Assembly asm = aNewDomain.Load("EnRegner");
            Type type = asm.GetType("EnRegner.EnRegner");
            object regner = asm.CreateInstance("EnRegner.EnRegner");

            MethodInfo mi = type.GetMethod("Plus");
            object[] argsToMethod = {40.4,45.6};


            object ores = mi.Invoke(regner, argsToMethod);

            Console.WriteLine("Result {0}", (double)ores);

              aNewDomain.DomainUnload += (o, s) =>
            {
                Console.WriteLine("Unloaded aNewDomain");
            };



            AppDomain.Unload(aNewDomain);




        }


    }
}
```


C:\WINDOWS\system32\cmd.exe

```
Result 86
Unloaded aNewDomain
```

Object Context

- As it can be seen above:
- AppDomains are logical partitions in a process!
- An AppDomain can furthermore be divided into contexts
  - Object context
- Every application has s default context (context 0)
  - The first created context
- It is possible to create an [Synchronization] object
- ”Synchronized object” (thereby **Thread-safe**)

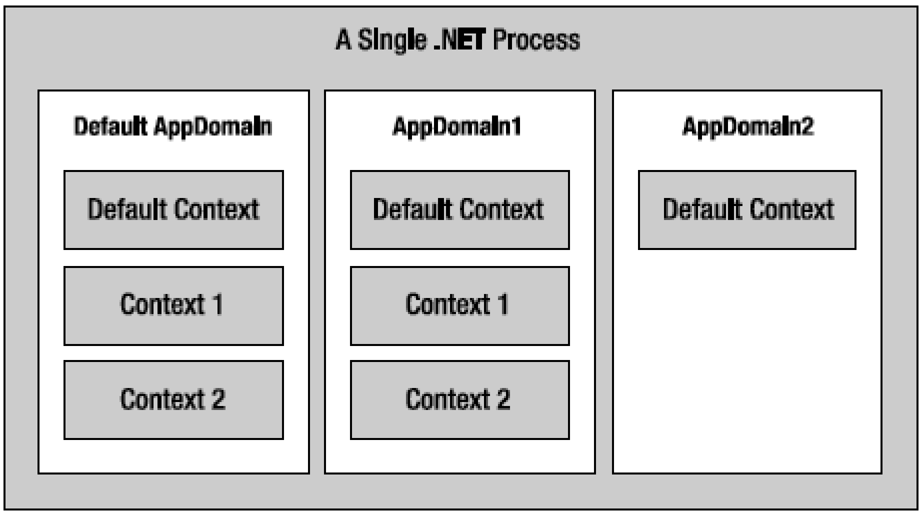
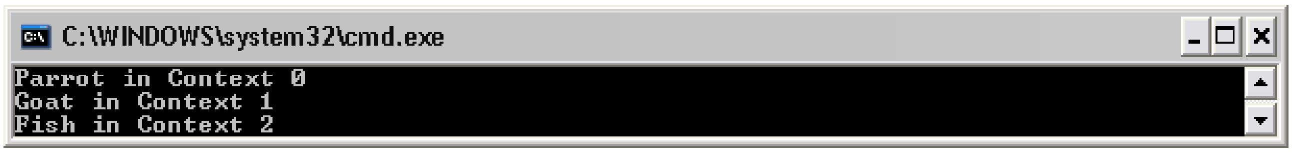


Figure 17-3. *Processes, application domains, and context boundaries*

To create object contexts

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Remoting.Contexts;
using System.Threading;

namespace ObjContext
{
    public class Parrot
    {
        public Parrot()
        {
            Context c = Thread.CurrentContext;
            Console.WriteLine("Parrot in Context {0}", c.ContextID);
        }
    }

    [Synchronization]
    public class Goat:ContextBoundObject
    {
        public Goat()
        {
            Context c = Thread.CurrentContext;
            Console.WriteLine("Goat in Context {0}", c.ContextID);
        }
    }

    [Synchronization]
    public class Fish : ContextBoundObject
    {
        public Fish()
        {
            Context c = Thread.CurrentContext;
            Console.WriteLine("Fish in Context {0}", c.ContextID);
        }
    }


    class Program
    {
        static void Main(string[] args)
        {

            Parrot p = new Parrot();

            Goat g = new Goat();

            Fish f = new Fish();

        }
    }
}
```

- two contexts are created (by attribute programming and inheritance)
- Parrot is still in "home" context



```
Parrot in Context 0
Goat in Context 1
Fish in Context 2
```

Multi threads and parallel programming

- There is no ”one to one” mapping of AppDomains and threads
    - AppDomains can have several threads and
    - one thread is not tied to an AppDomain

Threads works on cross of AppDomains and is managed bu the CLR scheduler

Concurrency problems

- No guarantee that a thread is started when the line of code is reached
    - CLR instructs the OS to start the thread
- You should know what parts of the application is thread-volatile (multi-threaded)
- Which operations are atomic (always safe in multi-threaded apps)
- Especially important if more threads work on the same data
- When (and which thread) changes data?

Synchronization

- As explained above we have just a little control on threads exact executing
    - When precisely is the thread started?
- Worse
    - Producer-consumer relations between threads

- .NET uses
    - lock
    - monitors
    - and synchronizations attributtes

Flashback on delegates

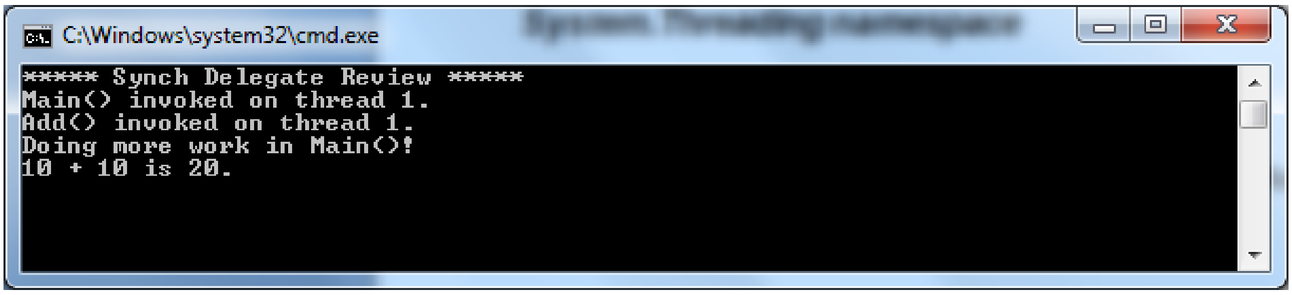
- Asynchron delegates

  - Creates it’s own thread

- First: Example- **synchronous** delegate

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AsyncDelegates
{
    class Program
    {
        public delegate int BinaryOp(int x, int y);

        static void Main(string[] args)
        {

            Console.WriteLine("***** Synch Delegate Review *****");
            // Print out the ID of the executing thread.
            Console.WriteLine("Main() invoked on thread {0}.",
            Thread.CurrentThread.ManagedThreadId);
            // Invoke Add() in a synchronous manner.
            BinaryOp b = new BinaryOp(Add);
            // Could also write b.Invoke(10, 10);
            int answer = b(10, 10);
            // These lines will not execute until
            // the Add() method has completed.
            Console.WriteLine("Doing more work in Main()!");
            Console.WriteLine("10 + 10 is {0}.", answer);
            Console.ReadLine();
        }

        static int Add(int x, int y)
        {
            // Print out the ID of the executing thread.
            Console.WriteLine("Add() invoked on thread {0}.",
            Thread.CurrentThread.ManagedThreadId);
            // Pause to simulate a lengthy operation.
            Thread.Sleep(5000);
            return x + y;
        }
    }
}
```

- Main thread waits for loong time (5s)

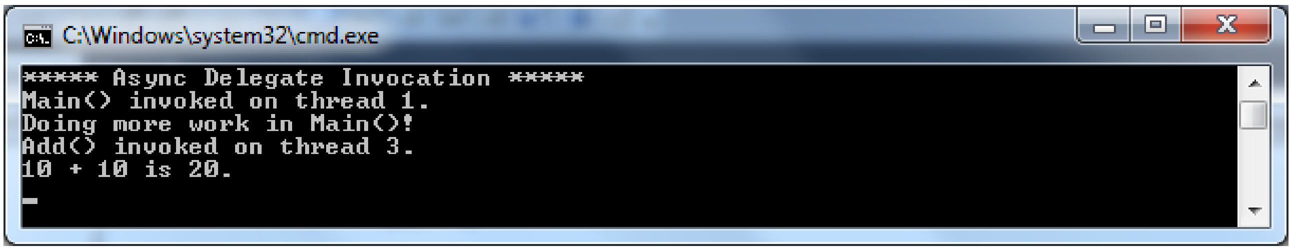


```
***** Synch Delegate Review *****
Main() invoked on thread 1.
Add() invoked on thread 1.
Doing more work in Main()!
10 + 10 is 20.
```

The Asynchrounous delegate

- BeginInvoke
- EndInvoke

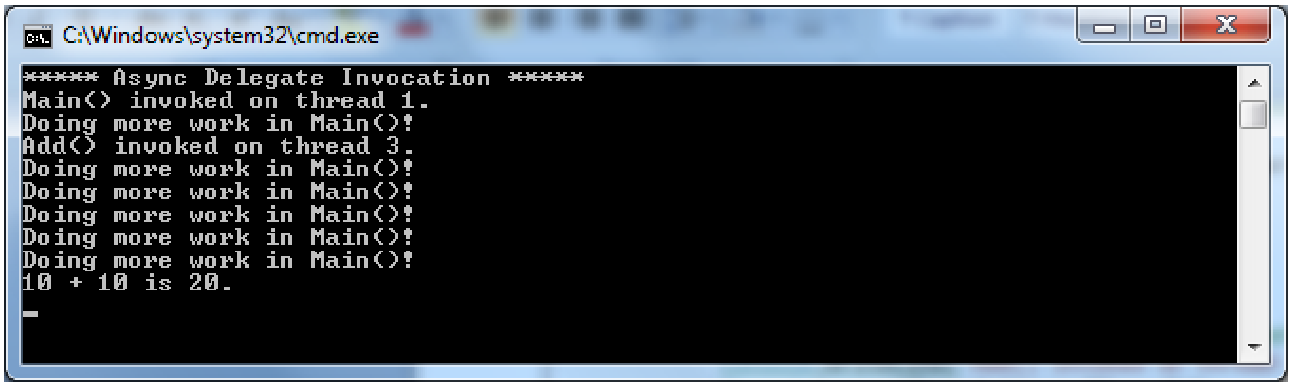
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AsyncDelegates
{
    class Program
    {
        public delegate int BinaryOp(int x, int y);

        static void Main(string[] args)
        {

            Console.WriteLine("***** Async Delegate Invocation *****");
            // Print out the ID of the executing thread.
            Console.WriteLine("Main() invoked on thread {0}.",
                                             Thread.CurrentThread.ManagedThreadId);
            // Invoke Add() on a secondary thread.
            BinaryOp b = new BinaryOp(Add);
            IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);
            // Do other work on primary thread...
            Console.WriteLine("Doing more work in Main()!");
            // Obtain the result of the Add()
            // method when ready.
            int answer = b.EndInvoke(iftAR);
            Console.WriteLine("10 + 10 is {0}.", answer);
            Console.ReadLine();
        }

        static int Add(int x, int y)
        {
            // Print out the ID of the executing thread.
            Console.WriteLine("Add() invoked on thread {0}.",
            Thread.CurrentThread.ManagedThreadId);
            // Pause to simulate a lengthy operation.
            Thread.Sleep(5000);
            return x + y;
        }
    }
}
```


C:\Windows\system32\cmd.exe

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
10 + 10 is 20.
_
```

- One problem:
  - Time betweenBeginInvoke and EndInvoke is clearly less than 5s
  - Can be solved in this way

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AsyncDelegates
{
    class Program
    {
        public delegate int BinaryOp(int x, int y);

        static void Main(string[] args)
        {

            Console.WriteLine("***** Async Delegate Invocation *****");
            // Print out the ID of the executing thread.
            Console.WriteLine("Main() invoked on thread {0}.",
                                          Thread.CurrentThread.ManagedThreadId);
            // Invoke Add() on a secondary thread.
            BinaryOp b = new BinaryOp(Add);
            IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);
            while (!iftAR.IsCompleted)
            {
                Console.WriteLine("Doing more work in Main()!");
                Thread.Sleep(1000);
            }
            // Now we know the Add() method is complete.
            int answer = b.EndInvoke(iftAR);
            Console.WriteLine("10 + 10 is {0}.", answer);
            Console.ReadLine();
        }

        static int Add(int x, int y)
        {
            // Print out the ID of the executing thread.
            Console.WriteLine("Add() invoked on thread {0}.",
            Thread.CurrentThread.ManagedThreadId);
            // Pause to simulate a lengthy operation.
            Thread.Sleep(5000);
            return x + y;
        }
    }
}
```



```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
10 + 10 is 20.
_
```

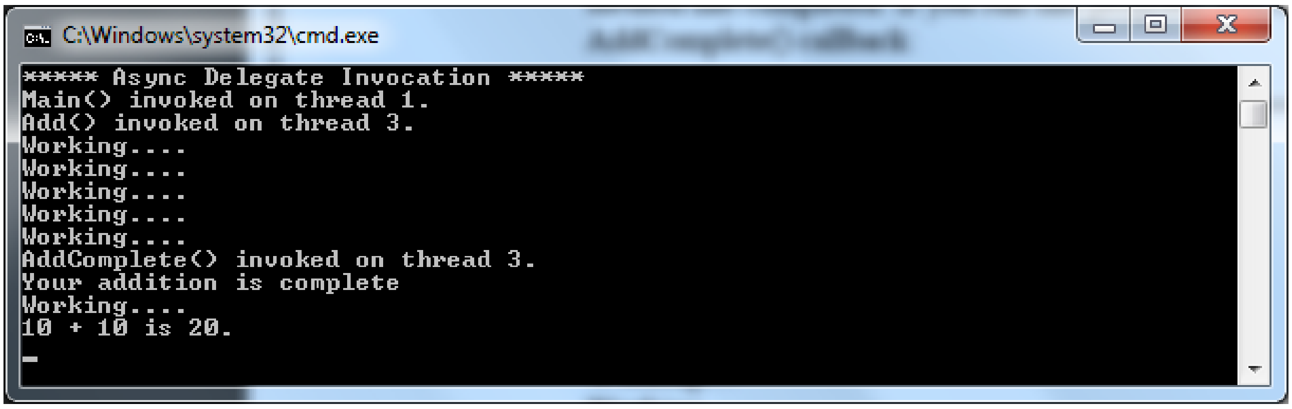
AsyncCallback delegates

Note The callback method will be called on the secondary thread, not the primary thread. This has important implications when using threads within a graphical user interface (WPF or Windows Forms) as controls have thread-affinity, meaning they can be manipulated only by the thread that created them. You'll see some examples of working the threads from a GUI later in this chapter, during the examination of the Task Parallel Library (TPL) and the new .NET 4.5 C# async and await keywords.

- Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace AsyncDelegates
{
    class Program
    {
        public delegate void BinaryOp(int x, int y);
        private static bool isDone = false;
        private static int answer;

        static void Main(string[] args)
        {

            Console.WriteLine("***** Async Delegate Invocation *****");
            // Print out the ID of the executing thread.
            Console.WriteLine("Main() invoked on thread {0}.",
                                               Thread.CurrentThread.ManagedThreadId);
            // Invoke Add() on a secondary thread.
            BinaryOp b = new BinaryOp(Add);
            IAsyncResult iftAR = b.BeginInvoke(10, 10,
                                new AsyncCallback(AddComplete), null);
            // Assume other work is performed here...
            while (!isDone)
            {
                Thread.Sleep(1000);
                Console.WriteLine("Working....");
            }
            Console.WriteLine("10 + 10 is {0}.", answer);
            Console.ReadLine();
        }

        static void Add(int x, int y)
        {
            // Print out the ID of the executing thread.
            Console.WriteLine("Add() invoked on thread {0}.",
            Thread.CurrentThread.ManagedThreadId);
            // Pause to simulate a lengthy operation.
            Thread.Sleep(5000);
            answer =  x + y;
        }


        static void AddComplete(IAsyncResult itfAR)
        {
            Console.WriteLine("AddComplete() invoked on thread {0}.",
           Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("Your addition is complete");
            isDone = true;
        }

    }
}
```

- AddComplete is called when data is ready
- `IAsyncResult iftAR = b.BeginInvoke(10, 10,`
- `                              new AsyncCallback(AddComplete), null);`

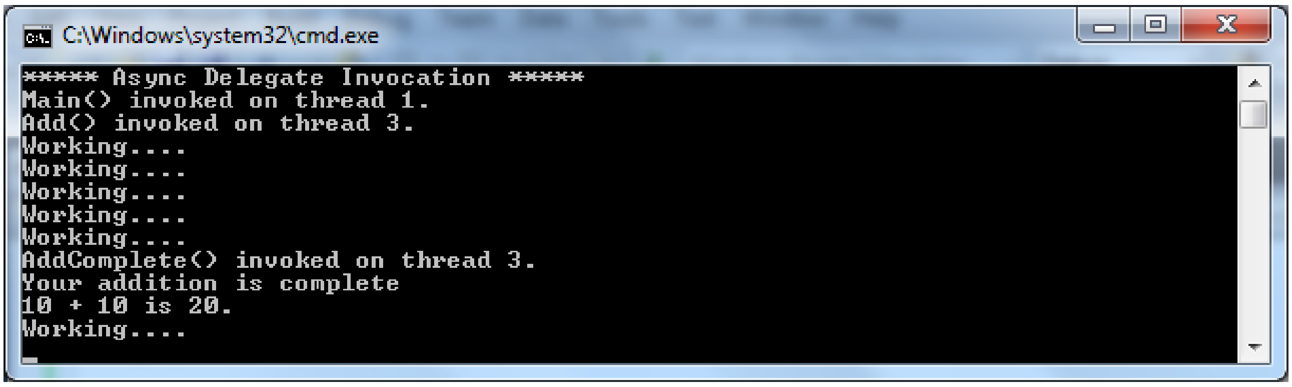


```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
Working....
10 + 10 is 20.
_
```

AsyncResult

- Instead of using private variable as above (and changing the delegate type returning void)
  - AsyncResult can be used
  - Remember: using `System.Runtime.Remoting.Messaging;`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Threading;
using System.Runtime.Remoting.Messaging;

namespace AsyncDelegates
{
    class Program
    {
        public delegate int BinaryOp(int x, int y);
        private static bool isDone = false;


        static void Main(string[] args)
        {

            Console.WriteLine("***** Async Delegate Invocation *****");
            // Print out the ID of the executing thread.
            Console.WriteLine("Main() invoked on thread {0}.",
                                               Thread.CurrentThread.ManagedThreadId);
            // Invoke Add() on a secondary thread.
            BinaryOp b = new BinaryOp(Add);
            IAsyncResult iftAR = b.BeginInvoke(10, 10,
                            new AsyncCallback(AddComplete), null);
            // Assume other work is performed here...
            while (!isDone)
            {
                Thread.Sleep(1000);
                Console.WriteLine("Working....");
            }

            Console.ReadLine();
        }

        static int Add(int x, int y)
        {
            // Print out the ID of the executing thread.
            Console.WriteLine("Add() invoked on thread {0}.",
            Thread.CurrentThread.ManagedThreadId);
            // Pause to simulate a lengthy operation.
            Thread.Sleep(5000);
            return x + y;
        }



        static void AddComplete(IAsyncResult itfAR)
        {
            Console.WriteLine("AddComplete() invoked on thread {0}.",
                                                 Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("Your addition is complete");
            // Now get the result.
            AsyncResult ar = (AsyncResult)itfAR;
            BinaryOp b = (BinaryOp)ar.AsyncDelegate;
            Console.WriteLine("10 + 10 is {0}.", b.EndInvoke(itfAR));
            isDone = true;
        }

    }
}
```


C:\Windows\system32\cmd.exe

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
10 + 10 is 20.
Working....
```

System.Threading namespace

- Contains
    - Interlocked (atomic op.)
    - Monitor (synchronization)
        - Uses locks / wait signals
    - Mutex
        - Synkronization between AppDomains
    - Semaphore
        - Reduce the number of thread's use of a ressource
    - Thread
    - ThreadPool (manage more threads)
    - TheadPriority

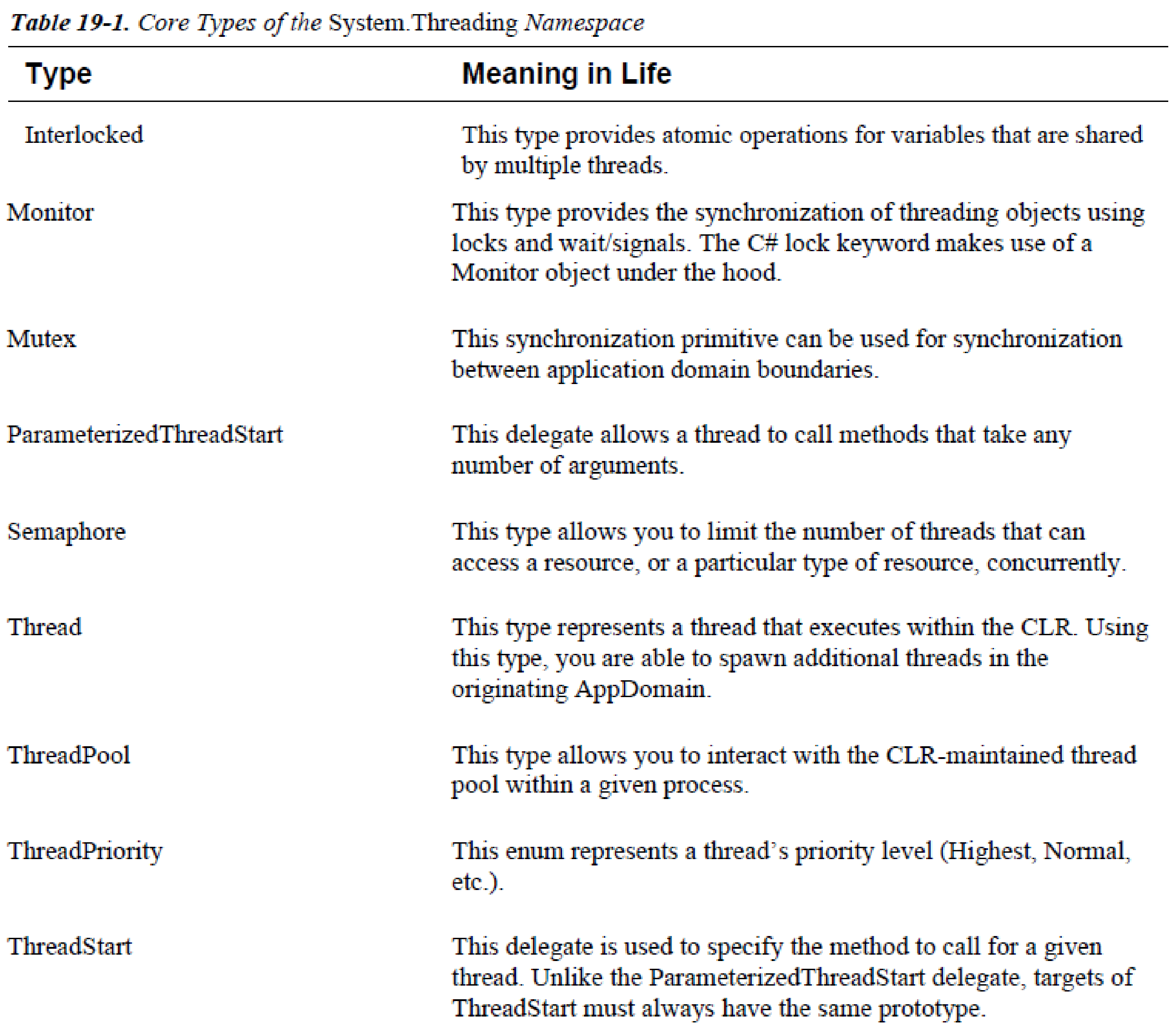


Table 19-1. *Core Types of the* System.Threading *Namespace*

Type	Meaning in Life
Interlocked	This type provides atomic operations for variables that are shared by multiple threads.
Monitor	This type provides the synchronization of threading objects using locks and wait/signals. The C# lock keyword makes use of a Monitor object under the hood.
Mutex	This synchronization primitive can be used for synchronization between application domain boundaries.
ParameterizedThreadStart	This delegate allows a thread to call methods that take any number of arguments.
Semaphore	This type allows you to limit the number of threads that can access a resource, or a particular type of resource, concurrently.
Thread	This type represents a thread that executes within the CLR. Using this type, you are able to spawn additional threads in the originating AppDomain.
ThreadPool	This type allows you to interact with the CLR-maintained thread pool within a given process.
ThreadPriority	This enum represents a thread's priority level (Highest, Normal, etc.).
ThreadStart	This delegate is used to specify the method to call for a given thread. Unlike the ParameterizedThreadStart delegate, targets of ThreadStart must always have the same prototype.

Thread class

- Can – SURPRISE – create new threads
  - Static members

Static Member	Meaning in Life
CurrentContext	This read-only property returns the context in which the thread is currently running.
CurrentThread	This read-only property returns a reference to the currently running thread.

GetDomain()
GetDomainID()
	These methods return a reference to the current AppDomain or the ID of the domain in which the current thread is running.

Sleep()	This method suspends the current thread for a specified time.

  - Instance members

Table 19-3. *Select Instance-Level Members of the Thread Type*

Instance-Level Member	Meaning in Life
IsAlive	Returns a Boolean that indicates whether this thread has been started (and has not yet terminated or aborted).
IsBackground	Gets or sets a value indicating whether or not this thread is a "background thread" (more details in just a moment).
Name	Allows you to establish a friendly text name of the thread.
Priority	Gets or sets the priority of a thread, which may be assigned a value from the ThreadPriority enumeration.
ThreadState	Gets the state of this thread, which may be assigned a value from the ThreadState enumeration.
Abort()	Instructs the CLR to terminate the thread as soon as possible.
Interrupt()	Interrupts (e.g., wakes) the current thread from a suitable wait period.
Join()	Blocks the calling thread until the specified thread (the one on which Join() is called) exits.
Resume()	Resumes a thread that has been previously suspended.
Start()	Instructs the CLR to execute the thread ASAP.
Suspend()	Suspends the thread. If the thread is already suspended, a call to Suspend() has no effect.

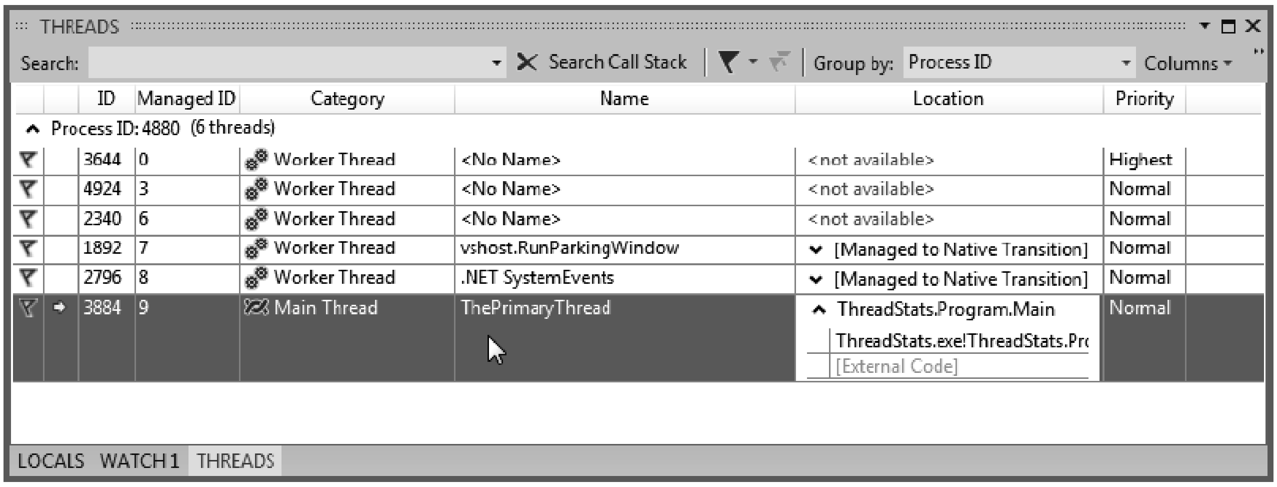
- A thread can have a name

Example:

```
using System;
using System.Threading;
using System.Diagnostics;


public class VisProcess
{


  public static void Main()

  {
         ArbejdsManden puha = new ArbejdsManden();
         Thread w;

         w = new Thread(new
                   ThreadStart(puha.KnokleKnokle));

           w.Start();
           w.Name= "Uno";
            w = new Thread(new
                      ThreadStart(puha.KnokleKnokle));
            // ThreadStart is a delegate

           w.Start();
           w.Name= "Duo";  // THERE IS ONLY ONE THREAD!





  }


}

class ArbejdsManden
{

    public void KnokleKnokle()
    {
         string who = string.Format("jeg hedder {0}",
                          Thread.CurrentThread.Name);
            while(true)
              Console.WriteLine(who);

        }

}
```

Output (partly) (random):

```
jeg hedder Uno
jeg hedder Uno
jeg hedder Duo
jeg hedder Duo
jeg hedder Duo
jeg hedder Duo
jeg hedder Duo
```



Process ID: 4880 (6 threads)

		ID	Managed ID	Category	Name	Location	Priority
		3644	0	Worker Thread	<No Name>	<not available>	Highest
		4924	3	Worker Thread	<No Name>	<not available>	Normal
		2340	6	Worker Thread	<No Name>	<not available>	Normal
		1892	7	Worker Thread	vshost.RunParkingWindow	[Managed to Native Transition]	Normal
		2796	8	Worker Thread	.NET SystemEvents	[Managed to Native Transition]	Normal
		3884	9	Main Thread	ThePrimaryThread	ThreadStats.Program.Main ThreadStats.exe!ThreadStats.Pr [External Code]	Normal

Figure 19-1. Debugging a thread with Visual Studio

Thread information

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Diagnostics;
using System.Threading;

namespace PogT
{
    class Program
    {
        static void Main(string[] args)
        {

            Console.WriteLine("***** Primary Thread stats *****\n");
            // Obtain and name the current thread.
            Thread primaryThread = Thread.CurrentThread;
            primaryThread.Name = "ThePrimaryThread";
            // Show details of hosting AppDomain/Context.
            Console.WriteLine("Name of current AppDomain: {0}",
            Thread.GetDomain().FriendlyName);
            Console.WriteLine("ID of current Context: {0}",
            Thread.CurrentContext.ContextID);
            // Print out some stats about this thread.
            Console.WriteLine("Thread Name: {0}",
            primaryThread.Name);
            Console.WriteLine("Has thread started?: {0}",
            primaryThread.IsAlive);
            Console.WriteLine("Priority Level: {0}",
            primaryThread.Priority);
            Console.WriteLine("Thread State: {0}",
            primaryThread.ThreadState);
            Console.ReadLine();


        }

    }
}
```

Thread priority

- Thread priority
  - Watch out for VERY OLD O.S.
- 5 levels
  - AboveNormal
  - BelowNormal
  - Highest
  - lowest
  - Normal (default)

- This is only a guideline
  - The Scheduler can be occupied by other work

```
...

 w = new Thread(new ThreadStart(puha.KnokleKnokle));
 w.Start();
 w.Name= "Duo";
 w.Priority=ThreadPriority.Highest;
...
```

To create a secundary thead

1. Create a method to be executed
2. Create a delegate (ThreadStart or ParameterizedThreadStart delegate)
3. Create a Thread object using the delegate
4. Set the initial conditions
5. Start the thread

Example: ThreadStart (delegate)

```
using System;
using System.Threading;
using System.Diagnostics;


public class NyVisProcess
{


  public static void Main()

  {
         Thread sek = new Thread(new
                       ThreadStart(EnTrådFunk));
         sek.Start(); // anmodning

         Console.ReadLine();

  }

  static void EnTrådFunk()
  {
        Thread.CurrentThread.Name ="Lokal tråd";
        Thread denne = Thread.CurrentThread;
        Console.WriteLine("Navn: {0}",denne.Name);

  }
}
```

- Start() requests CLR to start
- ThreadStart is a thread related delegate
    - Method can not have arguments

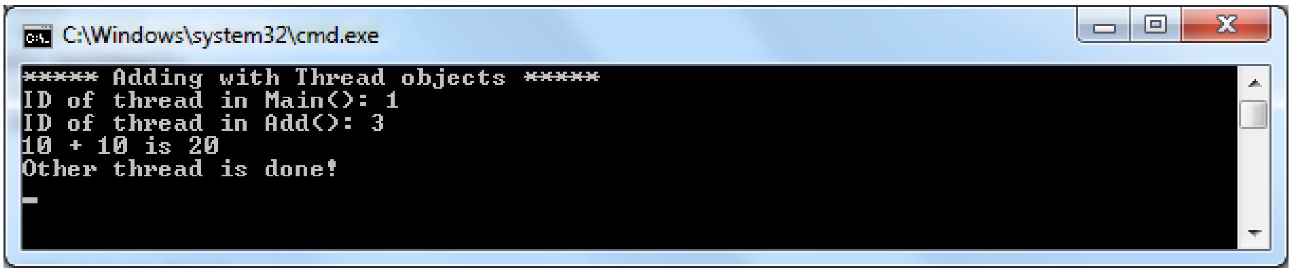
Example: ParameterizedThreadStart delegate

```
using System;
using System.Threading;

namespace ParamThread2
{
    class Program
    {
        static void Main(string[] args)
        {
            Program p = new Program();
            Thread arbejder1 = new Thread(new
                                   ParameterizedThreadStart(p.KnokleKnokle));
            Thread arbejder2 = new Thread(new
                                   ParameterizedThreadStart(p.KnokleKnokle));

            arbejder1.Start("Mette");
            arbejder2.Start(22);

        }

        public void KnokleKnokle(object data)
        {
            Console.WriteLine("Argument: {0} ", data);
        }
    }
}
```

AutoResetEvent class

- Information about when a thread is finished can be done in several ways
- One easy way:
- Define an object of AutoResetEvent (for example waitHandle)
- Let a thread wait until it is notified to start: waitHandle.WaitOne()
- Let another thread indform using waitHandle.Set()
- Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Diagnostics;
using System.Threading;

namespace PogT
{

    class AddParams
    {
        public int a, b;
        public AddParams(int numb1, int numb2)
        {
            a = numb1;
            b = numb2;
        }
    }
    class Program
    {
        private static AutoResetEvent waitHandle = new AutoResetEvent(false);

        static void Main(string[] args)
        {

            Console.WriteLine("***** Adding with Thread objects *****");
            Console.WriteLine("ID of thread in Main(): {0}",
            Thread.CurrentThread.ManagedThreadId);
            AddParams ap = new AddParams(10, 10);
            Thread t = new Thread(new ParameterizedThreadStart(Add));
            t.Start(ap);
            // Wait here until you are notified!
            waitHandle.WaitOne();
            Console.WriteLine("Other thread is done!");
            Console.ReadLine();


        }

        static void Add(object data)
        {
            if (data is AddParams)
            {
                Console.WriteLine("ID of thread in Add(): {0}",
                Thread.CurrentThread.ManagedThreadId);
                AddParams ap = (AddParams)data;
                Console.WriteLine("{0} + {1} is {2}",
                ap.a, ap.b, ap.a + ap.b);
                // Tell other thread we are done.
                waitHandle.Set();

            }
        }





    }
}
```



```
***** Adding with Thread objects *****
ID of thread in Main(): 1
ID of thread in Add(): 3
10 + 10 is 20
Other thread is done!
_
```

For- and background processes

- CLR divides a thread in two groups:
  - forground thread
    - critical tasks
    - can prevent an application to terminate
    - All threads mst be killed first
  - background thread
    - non critical tasks
    - daemon
    - Is stopped along with the application

A thread started with Thread.Start() is per definition:
A foreground thread

Creating a background thread

```
...
         Thread sek = new Thread(new
                       ThreadStart(EnTrådFunk));
         sek.IsBackground = true;

...
```

Suspension of threads

Sleep

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

Suspend

```
...
  w.Suspend();
  w.Resume();
...
```

Synchronization

- Common data
- Or Producer / consumer relation

Lock

```
using System;
using System.Threading;
using System.Diagnostics;


public class VisProcess
{


  public static void Main()

  {
         ArbejdsManden puha = new ArbejdsManden();
         Thread w;

           w = new Thread(new
                   ThreadStart(puha.KnokleKnokle));
           w.Start();
           w.Name= "Uno";
             w = new Thread(new
                    ThreadStart(puha.KnokleKnokle));
           w.Start();
           w.Name= "Duo";


           Console.ReadLine();

  }


}


class ArbejdsManden
{
    private int local;

    private object threadLock = new object();

    public void KnokleKnokle()
    {

        lock(threadLock)
        {
         local++;
         string who = string.Format("jeg hedder {0}",
                          Thread.CurrentThread.Name);
         string sValue ="Local:"+local.ToString();
         string msg = who + sValue;

         for(int i  = 0; i < 10;i++)
         {
           Console.WriteLine(msg);
           Thread.Sleep(1000);
         }
        }
    }

}
```

Output:

```
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder UnoLocal:1
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
jeg hedder DuoLocal:2
```

- If lock is used in a private method
    - can lock(this) be used

Monitor

- Lock is shorthand for Monitor
- **Uses Wait**
- **can inform waiting threads**
  - **uses Pulse() and PulseAll()**

Lock ”translated” to Monitor:

```
...
    public void KnokleKnokle()
    {

        Monitor.Enter(this);
        try
        {
          local++;
          string who = string.Format("jeg hedder {0}",
                          Thread.CurrentThread.Name);

          string sValue ="Local:"+local.ToString();
          string msg = who + sValue;

          for(int i  = 0; i < 10;i++)
          {
            Console.WriteLine(msg);
            Thread.Sleep(1000);
          }
        }
        finally
        {
           Monitor.Exit(this);
        }
    }
...
```

Work on atomic plan (System.Threading.Interlocked type)

Ensures that the operation is not interrupted

- Can increment
- decrement
- Exchange (swap two values)
- CompareExchange

Table 19-4. *Select Static Members of the System.Threading.Interlocked Type*

Member	Meaning in Life
CompareExchange()	Safely tests two values for equality and, if equal, exchanges one of the values with a third
Decrement()	Safely decrements a value by 1
Exchange()	Safely swaps two values
Increment()	Safely increments a value by 1

Incrementing

```
…
public void AddOne()
{
                lock(myLockToken)
                {
                   intVal++;
                }
}
```

Eller:

```
public void AddOne()
{
               int newVal = Interlocked.Increment(ref intVal);
}


…
```

Exchange

```
…
public void SafeAssignment()
```

```
{
                Interlocked.Exchange(ref myInt, 83);
}


…
```

The [Synchronization] attribute

```
…
using System.Runtime.Remoting.Contexts;
...
// All methods of Printer are now thread safe!
[Synchronization]
public class Printer : ContextBoundObject
{
public void PrintNumbers()
{
...
}

…
```

Timer Callbacks

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;


namespace Timer
{
    class Program
    {
        static void Main(string[] args)
        {
            Program p = new Program();
           TimerCallback æggeur = new TimerCallback(p.Koger);
           Timer t =
                new Timer(æggeur, null, 0, 5000);
           Console.WriteLine("Press enter to stop....");
           Console.ReadLine();
        }

        public void Koger(object state)
        {
            Console.WriteLine("Boiled in 5 sec.");
            }
        }
}
```

- Notice:
    - can NOT be used for time critical tasks

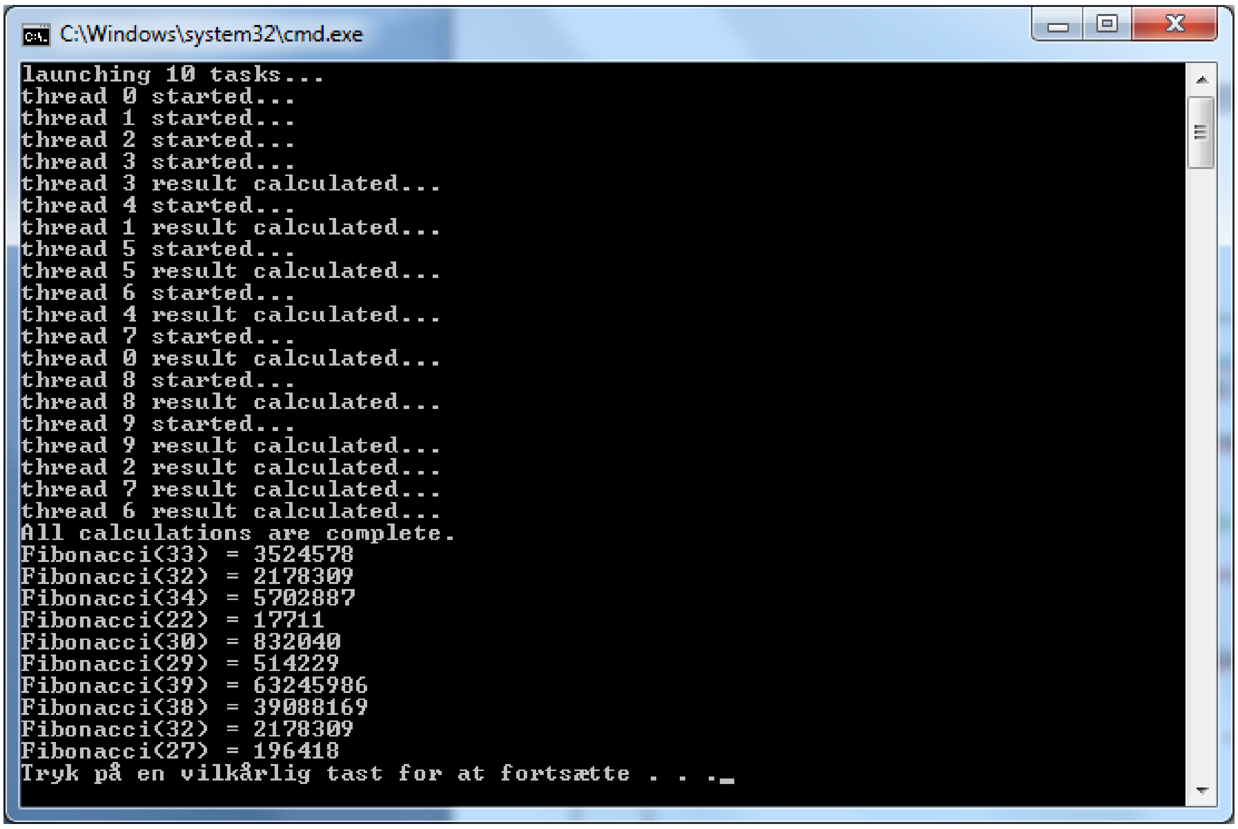
```
Timer t = new Timer(æggeur, null, 0, 5000);
```

null: No further arguments to the method (only object state)

ThreadPool

- A ThreadPool manage a number of threads effectively
- Easy to use
- Disadvantage:
    - Threads are always background threads
    - Priority can not be set
    - Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace ThreadPoolEx
{
    public class Fibonacci
    {
        public Fibonacci(int n, ManualResetEvent doneEvent)
        {
            _n = n;
            _doneEvent = doneEvent;
        }
        // Wrapper method for use with thread pool.
        public void ThreadPoolCallback(Object threadContext)
        {
            int threadIndex = (int)threadContext;
            Console.WriteLine("thread {0} started...", threadIndex);
            _fibOfN = Calculate(_n);
            Console.WriteLine("thread {0} result calculated...", threadIndex);
            _doneEvent.Set();
        }

        // Recursive method that calculates the Nth Fibonacci number.
        public int Calculate(int n)
        {
            if (n <= 1)
            {
                return n;
            }

            return Calculate(n - 1) + Calculate(n - 2);
        }

        public int N { get { return _n; } }
        private int _n;

        public int FibOfN { get { return _fibOfN; } }
        private int _fibOfN;

        private ManualResetEvent _doneEvent;
    }

    public class ThreadPoolExample
    {
        static void Main()
        {
            const int FibonacciCalculations = 10;

            // One event is used for each Fibonacci object
            ManualResetEvent[] doneEvents = new ManualResetEvent[FibonacciCalculations];
            Fibonacci[] fibArray = new Fibonacci[FibonacciCalculations];
            Random r = new Random();

            // Configure and launch threads using ThreadPool:
            Console.WriteLine("launching {0} tasks...", FibonacciCalculations);
            for (int i = 0; i < FibonacciCalculations; i++)
            {
                doneEvents[i] = new ManualResetEvent(false);
                Fibonacci f = new Fibonacci(r.Next(20, 40), doneEvents[i]);
                fibArray[i] = f;
                ThreadPool.QueueUserWorkItem(f.ThreadPoolCallback, i);
            }

            // Wait for all threads in pool to calculation...
            WaitHandle.WaitAll(doneEvents);
            Console.WriteLine("All calculations are complete.");

            // Display the results...
            for (int i = 0; i < FibonacciCalculations; i++)
            {
                Fibonacci f = fibArray[i];
                Console.WriteLine("Fibonacci({0}) = {1}", f.N, f.FibOfN);
            }
        }
    }

}

// Source: msdn
```



```
launching 10 tasks...
thread 0 started...
thread 1 started...
thread 2 started...
thread 3 started...
thread 3 result calculated...
thread 4 started...
thread 1 result calculated...
thread 5 started...
thread 5 result calculated...
thread 6 started...
thread 4 result calculated...
thread 7 started...
thread 0 result calculated...
thread 8 started...
thread 8 result calculated...
thread 9 started...
thread 9 result calculated...
thread 2 result calculated...
thread 7 result calculated...
thread 6 result calculated...
All calculations are complete.
Fibonacci(33) = 3524578
Fibonacci(32) = 2178309
Fibonacci(34) = 5702887
Fibonacci(22) = 17711
Fibonacci(30) = 832040
Fibonacci(29) = 514229
Fibonacci(39) = 63245986
Fibonacci(38) = 39088169
Fibonacci(32) = 2178309
Fibonacci(27) = 196418
Tryk på en vilkårlig tast for at fortsætte . . .
```

TPL (Task Parallel Library)

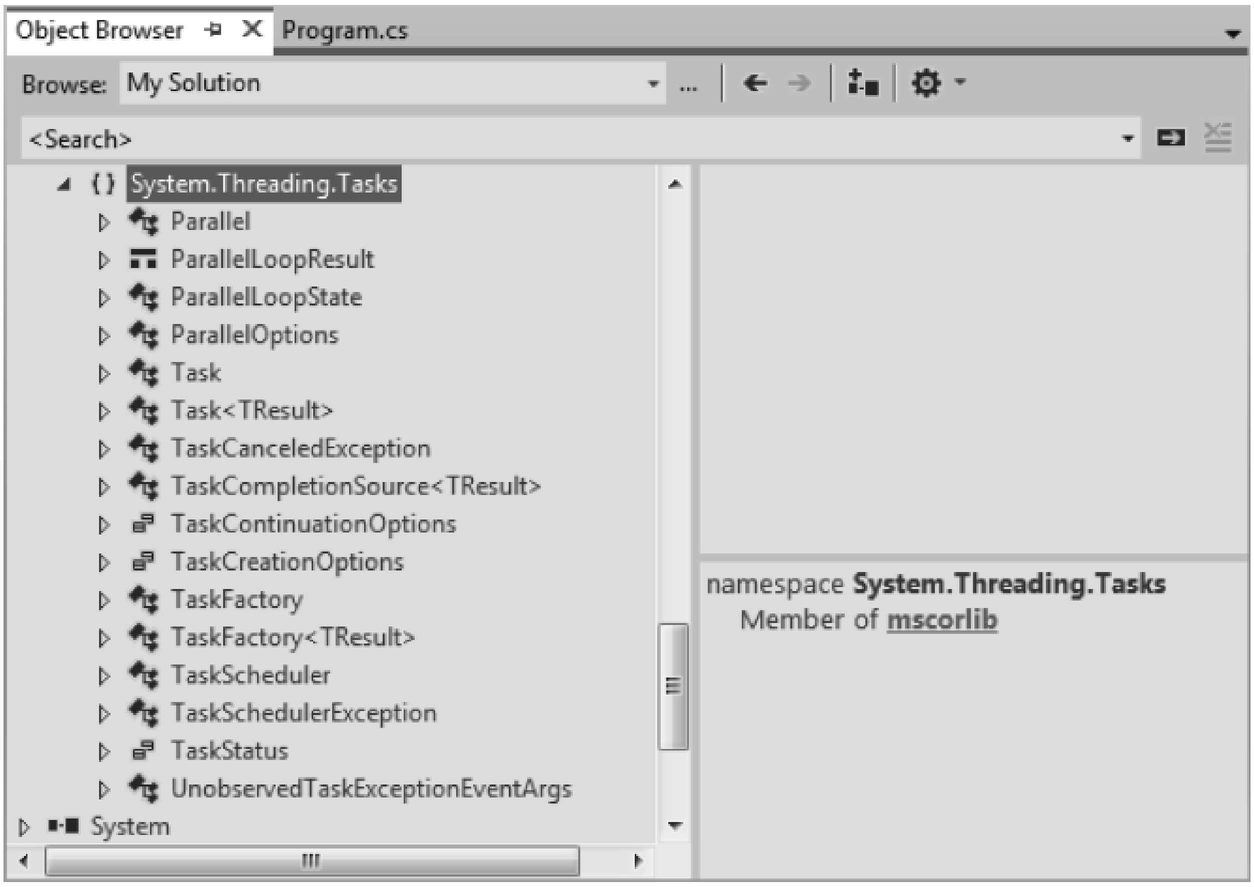
Parallel programming

- New since the 4.0 platform
- Used in machines withe more CPU's (cores)
- Model of the futures
- Uses System.Threading.Tasks

System.Threading.Tasks namespace

- Task Parallel Library (TPL)
    - Will automatically divide workload on CPU's using CLR ThreadPool

- Normal threads can of course still be used
- Many Parallel Tasks can create bad performance


Object Browser
Program.cs
Browse: My Solution
<Search>

```
{ } System.Threading.Tasks
    Parallel
    ParallelLoopResult
    ParallelLoopState
    ParallelOptions
    Task
    Task<TResult>
    TaskCanceledException
    TaskCompletionSource<TResult>
    TaskContinuationOptions
    TaskCreationOptions
    TaskFactory
    TaskFactory<TResult>
    TaskScheduler
    TaskSchedulerException
    TaskStatus
    UnobservedTaskExceptionEventArgs
System
```

namespace **System.Threading.Tasks**
Member of **mscorlib**

Figure 19-3. *Members of the System.Threading.Tasks namespace*

System.Threading.Tasks.Parallel class

- two primary methods
    - Parallel.For()
    - Parallel.ForEach()
    - Divides workload in a Parallel principle

    - Example with blocking (blocks while the it is working)

```
...
private void ProcessFiles()
{
// Load up all *.jpg files, and make a new folder for the modified data.
string[] files = Directory.GetFiles
(@"C:\Users\Public\Pictures\Sample Pictures", "*.jpg",
SearchOption.AllDirectories);
string newDir = @"C:\ModifiedPictures";
```

```
Directory.CreateDirectory(newDir);
// Process the image data in a blocking manner.
foreach (string currentFile in files)
{
string filename = Path.GetFileName(currentFile);
using (Bitmap bitmap = new Bitmap(currentFile))
{
bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
bitmap.Save(Path.Combine(newDir, filename));
// Print out the ID of the thread processing the current image.
this.Text = string.Format("Processing {0} on thread {1}", filename,
Thread.CurrentThread.ManagedThreadId);
}


…
```

- SOLUTION with parallel processes (Parallel.ForEach())

```
…
// Process the image data in a parallel manner!
Parallel.ForEach(files, currentFile =>
{
string filename = Path.GetFileName(currentFile);
using (Bitmap bitmap = new Bitmap(currentFile))
{
bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
bitmap.Save(Path.Combine(newDir, filename));
// This code statement is now a problem! See next section.
// this.Text = string.Format("Processing {0} on thread {1}", filename,
// Thread.CurrentThread.ManagedThreadId);
}
}
);


…
```

- Will now use as many CPU's as possible

WPF: Accesing UI elements on Secondary Threads (Dispatcher)

```
using (Bitmap bitmap = new Bitmap(currentFile))
{
  bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
  bitmap.Save(Path.Combine(outputDirectory, filename));

  // Eek! This will not work anymore!
  //this.Title = $"Processing {filename} on thread {Thread.CurrentThread.ManagedThreadId}";

  // Invoke on the Form object, to allow secondary threads to access controls
  // in a thread-safe manner.
  Dispatcher?.Invoke(() =>
  {
    this.Title =
     $"Processing {filename} on thread {Thread.CurrentThread.ManagedThreadId}";
  }
  );
}
```

Task class

- System.Threading.Tasks namespace contains a Task class
- more easy to use than an asynchron delegate
- Has a Task.Factory.StartNew()

```
..
private void btnProcessImages_Click(object sender, EventArgs e)
{
// Start a new "task" to process the files.
Task.Factory.StartNew(() =>
{
     ProcessFiles();
});
}


...
```

CancellationToken - Tasks

- Use a ParallelOptions object containing a CancellationTokenSource object
- If the user press Cancel All threads execution will be stopped
- See page 558

About Task parallelization

- See example in the book:
  - book is fetched using DownloadStringAsync(..)

```
…
static void GetBook()
{
WebClient wc = new WebClient();
wc.DownloadStringCompleted += (s, eArgs) =>
{
theEBook = eArgs.Result;
txtBook.Text = theEBook;
};
// The Project Gutenberg EBook of A Tale of Two Cities, by Charles Dickens
wc.DownloadStringAsync(new Uri("http://www.gutenberg.org/files/98/98-8.txt"));
}


…
```

  - and in parallel
  - Two time consuming tasks:
    - FindTenMostCommon(..)
    - FindLongestWord(..)

```
…
private string[] FindTenMostCommon(string[] words)
{
var frequencyOrder = from word in words
                     where word.Length > 6
                     group word by word into g
                     orderby g.Count() descending
                     select g.Key;
string[] commonWords = (frequencyOrder.Take(10)).ToArray();
return commonWords;
}
```

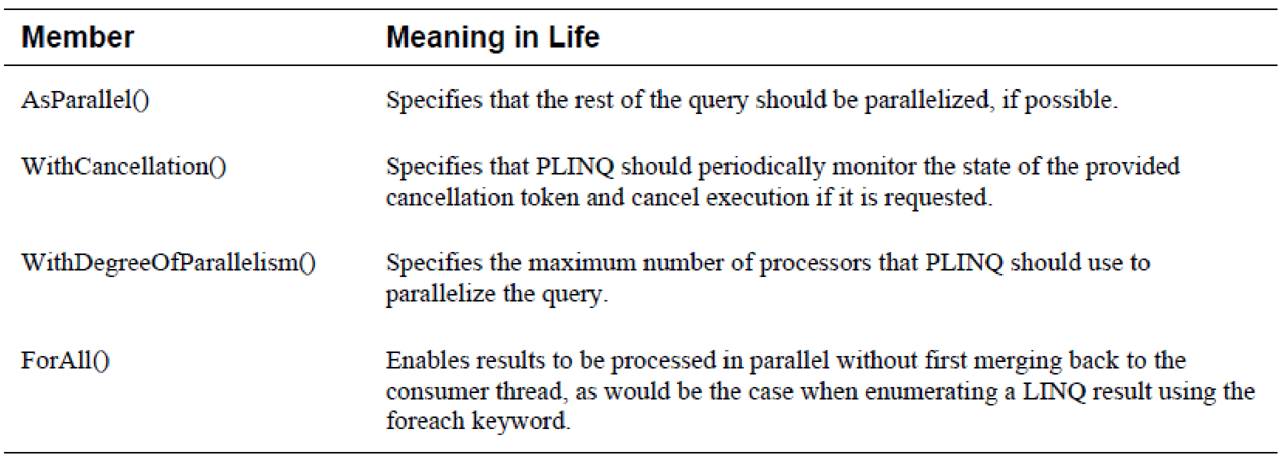
```
...

private string FindLongestWord(string[] words)
{
return (from w in words orderby w.Length descending select w).FirstOrDefault();
}

...
```

- Notice the use of LINQ

Parallel LINQ (PLINQ)

- It is possible to use extension methods allow the work to be done in parallel
- On run-time PLINQ analyze if parallel execution can be useful

Table 19-5. *Select Members of the ParallelEnumerable class*



Member	Meaning in Life
AsParallel()	Specifies that the rest of the query should be parallelized, if possible.
WithCancellation()	Specifies that PLINQ should periodically monitor the state of the provided cancellation token and cancel execution if it is requested.
WithDegreeOfParallelism()	Specifies the maximum number of processors that PLINQ should use to parallelize the query.
ForAll()	Enables results to be processed in parallel without first merging back to the consumer thread, as would be the case when enumerating a LINQ result using the foreach keyword.

- See example using AsParallel:

```
..

int[] modThreeIsZero = (from num in source.AsParallel() where num % 3 == 0
orderby num descending select num).ToArray();

..
```

- See example using AsParallel().WithCancellation

```
…
private void ProcessIntData()
{
// Get a very large array of integers.
int[] source = Enumerable.Range(1, 10000000).ToArray();
// Find the numbers where num % 3 == 0 is true, returned
// in descending order.
int[] modThreeIsZero = null;
try
{
       modThreeIsZero = (from num in
       source.AsParallel().WithCancellation(cancelToken.Token)
       where num % 3 == 0 orderby num descending
       select num).ToArray();
         MessageBox.Show(string.Format("Found {0} numbers that match query!",
         modThreeIsZero.Count()));
}
catch (OperationCanceledException ex)
{

}
}
…
```

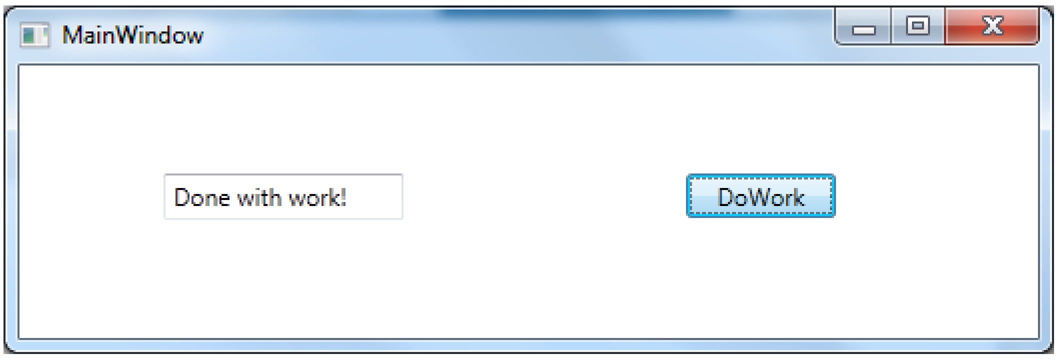
async call (since platform 4.5) (Updated 7.0 / 7.1)

- Two keywords
  - async
  - await

- Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;



using System.Threading;
```

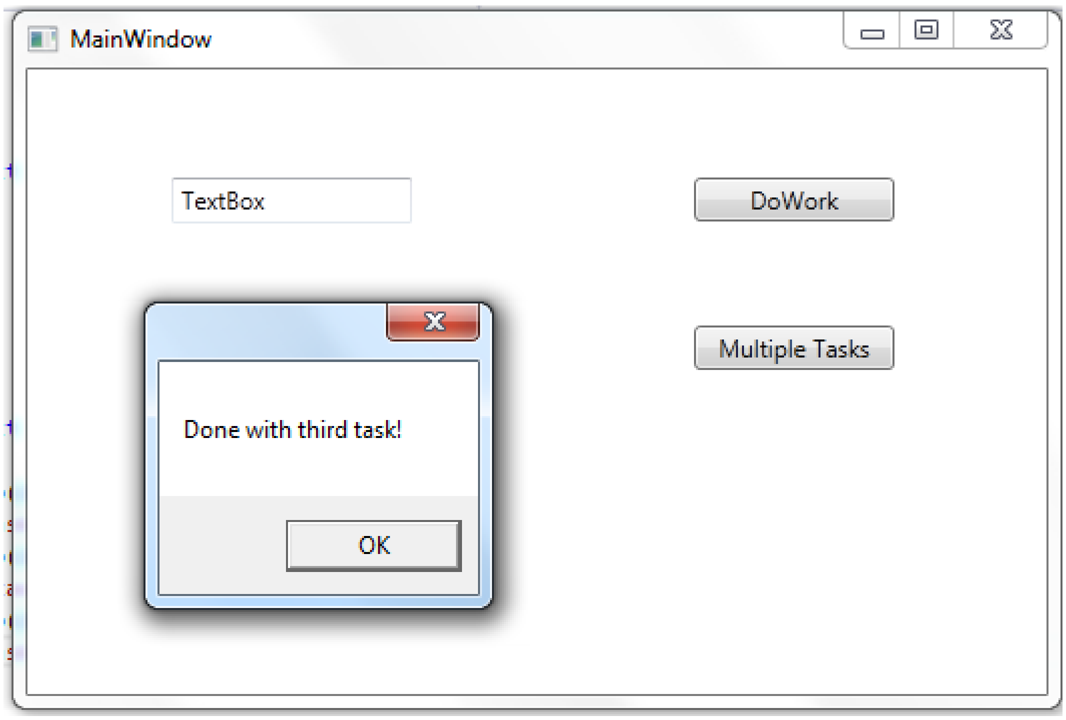
```
namespace AsyncAwaitExample
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }



        private Task<string> DoWork()
        {
            return Task.Run(() =>
            {
                Thread.Sleep(5000);
                return "Done with work!";
            });
        }

        private async void Button_Click_1(object sender, RoutedEventArgs e)
        {
            textBox1.Clear();

            textBox1.Text = await DoWork();

        }

    }
}
```



Note An "awaitable" method is simply a method that returns a Task<T>.

- an async method **has always one or more** await call

```
…

private async void Button_Click_2(object sender, RoutedEventArgs e)
        {
            await Task.Run(() => { Thread.Sleep(2000); });
            MessageBox.Show("Done with first task!");
            await Task.Run(() => { Thread.Sleep(2000); });
            MessageBox.Show("Done with second task!");
            await Task.Run(() => { Thread.Sleep(2000); });
            MessageBox.Show("Done with third task!");
        }
…
```



Done with third task!

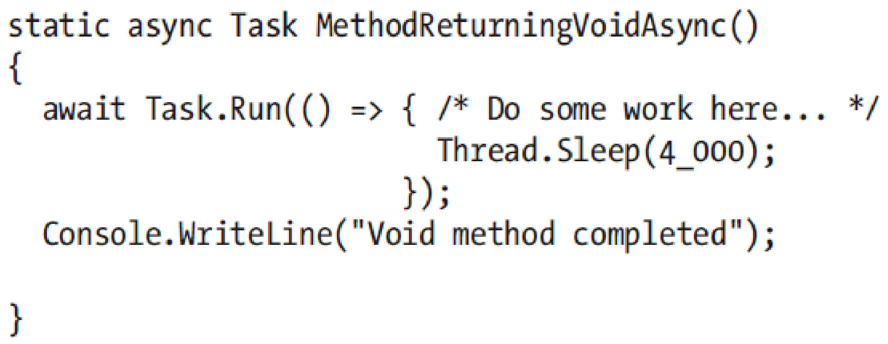
- Information
- Methods can be marked with async (also lambda expressions and anonymous methods)
- Methods are running ”blocking” mode until the await call – hereafter non-blocking (threaded)
- An async method can have several await

- Main can be declared async

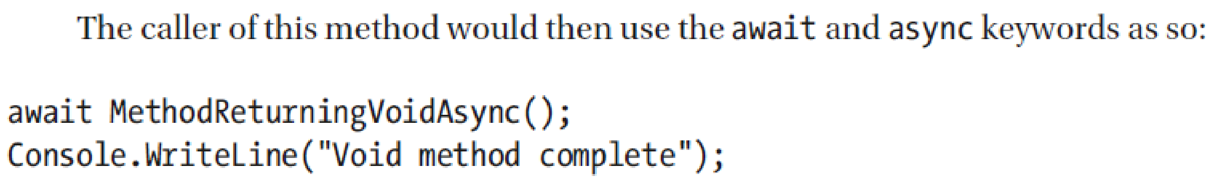
```
static async Task Main(string[] args)
{
  //ommitted for brevity
  string message = await DoWorkAsync();
  Console.WriteLine(message);
  //ommitted for brevity
}

static string DoWork()
{
  Thread.Sleep(5_000);
  return "Done with work!";
}
static async Task<string> DoWorkAsync()
{
  return await Task.Run(() =>
  {
    Thread.Sleep(5_000);
    return "Done with work!";
  });
}
```

- Asynchronous methods must always use the await keyword

Async methods returning void



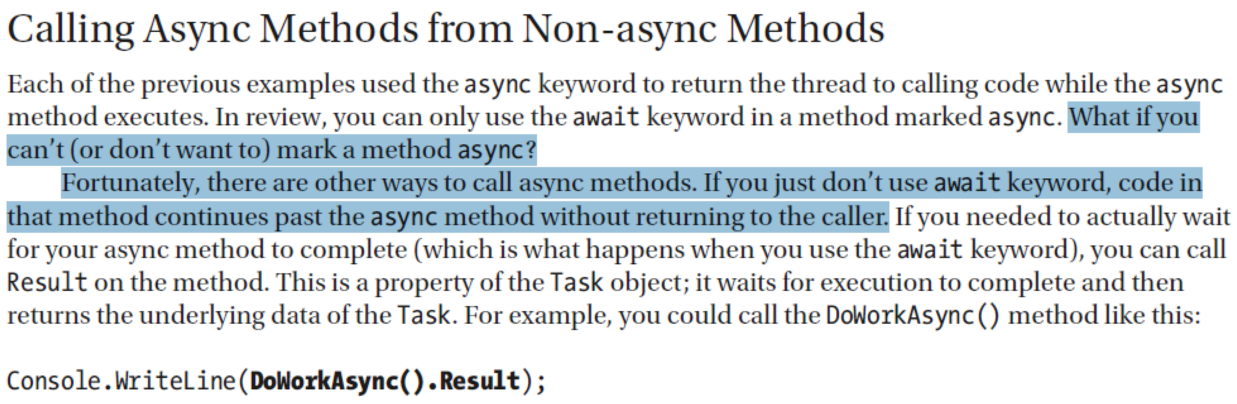
```
static async Task MethodReturningVoidAsync()
{
  await Task.Run(() => { /* Do some work here... */
                         Thread.Sleep(4_000);
                       });
  Console.WriteLine("Void method completed");

}
```



The caller of this method would then use the `await` and `async` keywords as so:

```
await MethodReturningVoidAsync();
Console.WriteLine("Void method complete");
```

Calling async methods from non-async methods



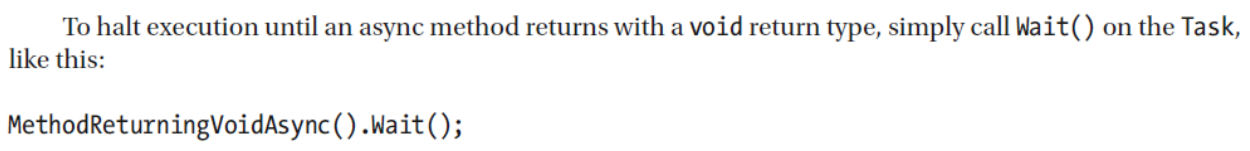
Calling Async Methods from Non-async Methods

Each of the previous examples used the `async` keyword to return the thread to calling code while the `async` method executes. In review, you can only use the `await` keyword in a method marked `async`. What if you can't (or don't want to) mark a method `async`?

Fortunately, there are other ways to call async methods. If you just don't use `await` keyword, code in that method continues past the `async` method without returning to the caller. If you needed to actually wait for your async method to complete (which is what happens when you use the `await` keyword), you can call `Result` on the method. This is a property of the `Task` object; it waits for execution to complete and then returns the underlying data of the `Task`. For example, you could call the `DoWorkAsync()` method like this:

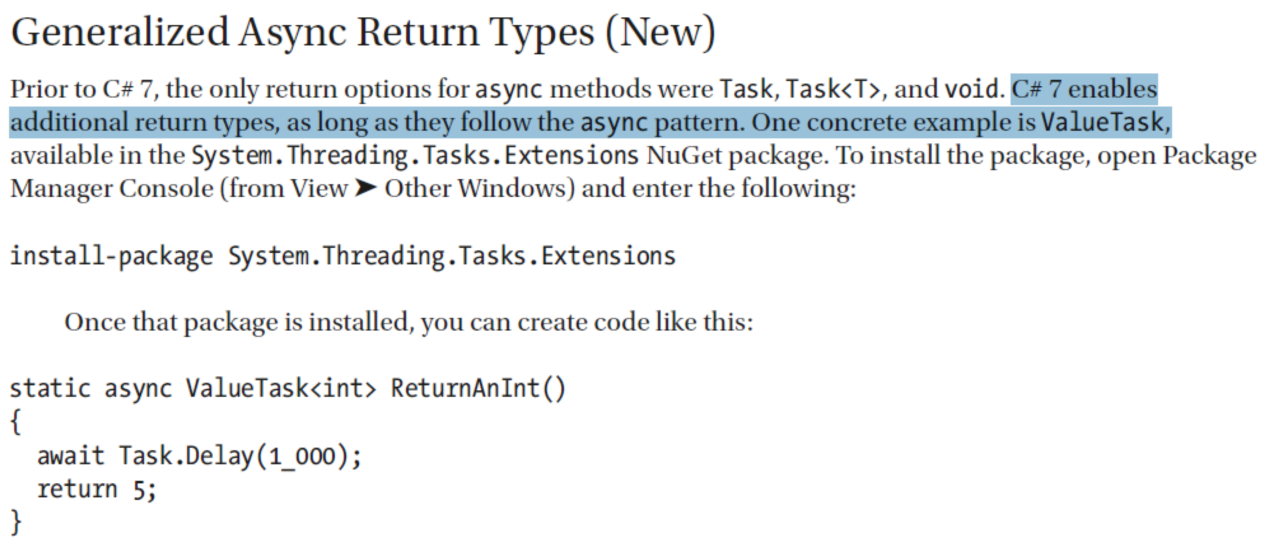
```
Console.WriteLine(DoWorkAsync().Result);
```

- Notice .Result


To halt execution until an async method returns with a `void` return type, simply call `Wait()` on the `Task`, like this:

```
MethodReturningVoidAsync().Wait();
```

Generalized Async Return Types (C# 7) – ValueTask< >


Generalized Async Return Types (New)

Prior to C# 7, the only return options for `async` methods were `Task`, `Task<T>`, and `void`. C# 7 enables additional return types, as long as they follow the `async` pattern. One concrete example is `ValueTask`, available in the `System.Threading.Tasks.Extensions` NuGet package. To install the package, open Package Manager Console (from View ➤ Other Windows) and enter the following:

```
install-package System.Threading.Tasks.Extensions
```

Once that package is installed, you can create code like this:

```
static async ValueTask<int> ReturnAnInt()
{
  await Task.Delay(1_000);
  return 5;
}
```

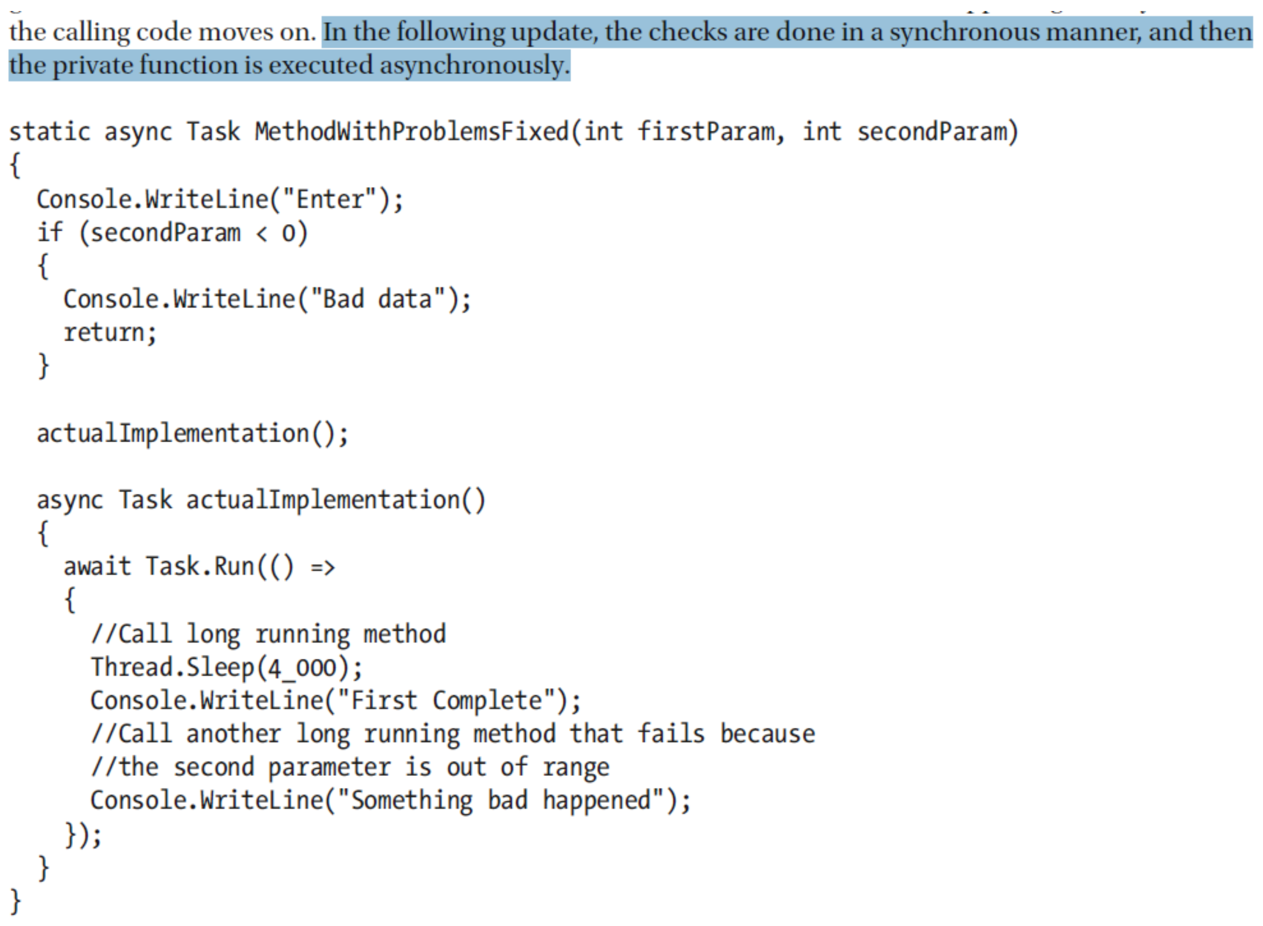
Local Functions (C# 7)

Problem:

```
static async Task MethodWithProblems(int firstParam, int secondParam)
{
  Console.WriteLine("Enter");
  await Task.Run(() =>
  {
    //Call long running method
    Thread.Sleep(4_000);
    Console.WriteLine("First Complete");
    //Call another long running method that fails because
    //the second parameter is out of range
    Console.WriteLine("Something bad happened");
  });
}
```

The scenario is that the second long-running task fails because of invalid input data. You can (and should) add checks to the beginning of the method, but since the entire method is asynchronous, there's no guarantee when the checks will be executed. It would be better for the checks to happen right away before the calling code moves on. In the following update, the checks are done in a synchronous manner, and then the private function is executed asynchronously.

Solution:

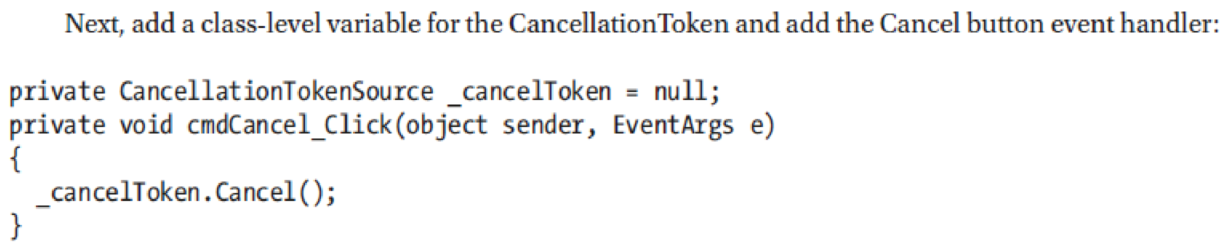


the calling code moves on. In the following update, the checks are done in a synchronous manner, and then the private function is executed asynchronously.

```
static async Task MethodWithProblemsFixed(int firstParam, int secondParam)
{
  Console.WriteLine("Enter");
  if (secondParam < 0)
  {
    Console.WriteLine("Bad data");
    return;
  }

  actualImplementation();

  async Task actualImplementation()
  {
    await Task.Run(() =>
    {
      //Call long running method
      Thread.Sleep(4_000);
      Console.WriteLine("First Complete");
      //Call another long running method that fails because
      //the second parameter is out of range
      Console.WriteLine("Something bad happened");
    });
  }
}
```

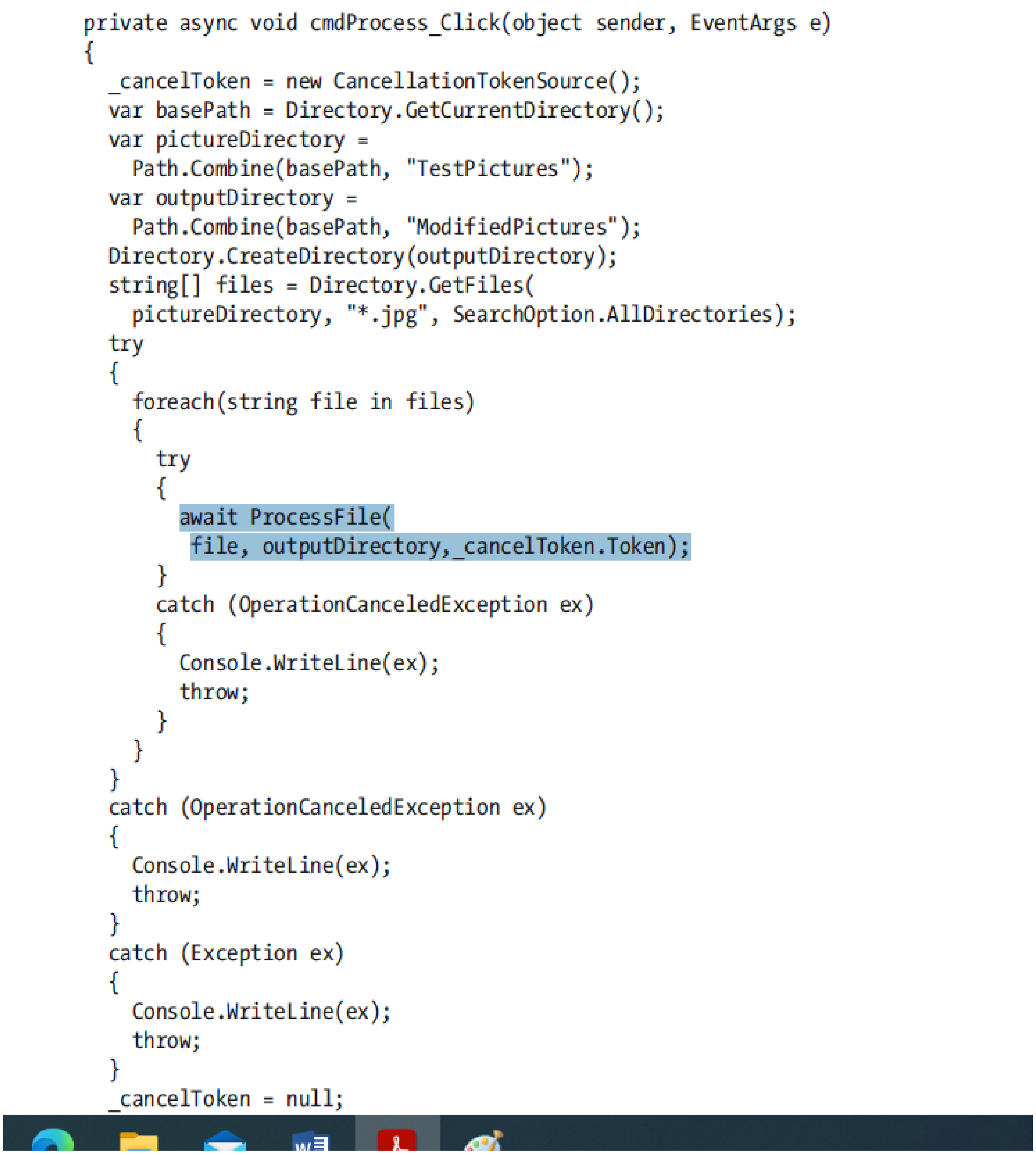
Cancelling async – await operations

- Cancel event



Next, add a class-level variable for the CancellationToken and add the Cancel button event handler:

```
private CancellationTokenSource _cancelToken = null;
private void cmdCancel_Click(object sender, EventArgs e)
{
  _cancelToken.Cancel();
}
```



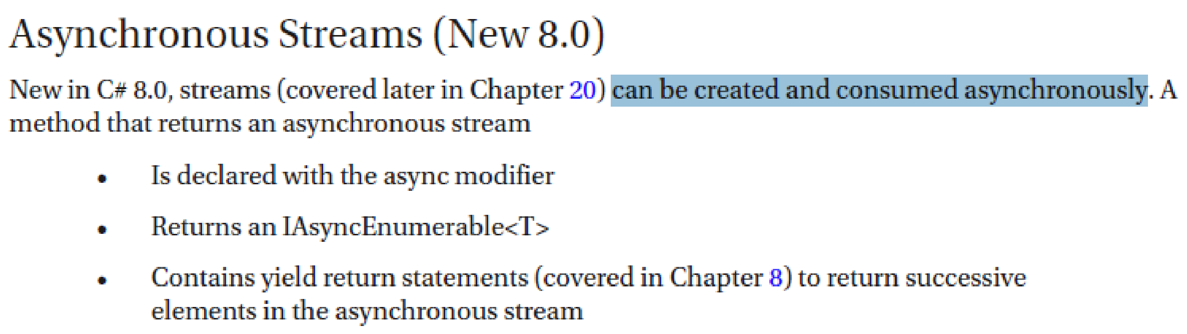
```
private async void cmdProcess_Click(object sender, EventArgs e)
{
  _cancelToken = new CancellationTokenSource();
  var basePath = Directory.GetCurrentDirectory();
  var pictureDirectory =
    Path.Combine(basePath, "TestPictures");
  var outputDirectory =
    Path.Combine(basePath, "ModifiedPictures");
  Directory.CreateDirectory(outputDirectory);
  string[] files = Directory.GetFiles(
    pictureDirectory, "*.jpg", SearchOption.AllDirectories);
  try
  {
    foreach(string file in files)
    {
      try
      {
        await ProcessFile(
         file, outputDirectory,_cancelToken.Token);
      }
      catch (OperationCanceledException ex)
      {
        Console.WriteLine(ex);
        throw;
      }
    }
  }
  catch (OperationCanceledException ex)
  {
    Console.WriteLine(ex);
    throw;
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
    throw;
  }
  _cancelToken = null;
```

- Final method to add:

The final method to add is the ProcessFile method.

```
private async Task ProcessFile(string currentFile,
  string outputDirectory, CancellationToken token)
{
  string filename = Path.GetFileName(currentFile);
  using (Bitmap bitmap = new Bitmap(currentFile))
  {
    try
    {
      await Task.Run(() =>
      {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(
            Path.Combine(outputDirectory, filename));
        Dispatcher?.Invoke(() =>
        {
          this.Title = $"Processing {filename}";
        });
      }
      ,token);
    }
    catch (OperationCanceledException ex)
    {
      Console.WriteLine(ex);
      throw;
    }
  }
}
```

Asynchronous streams

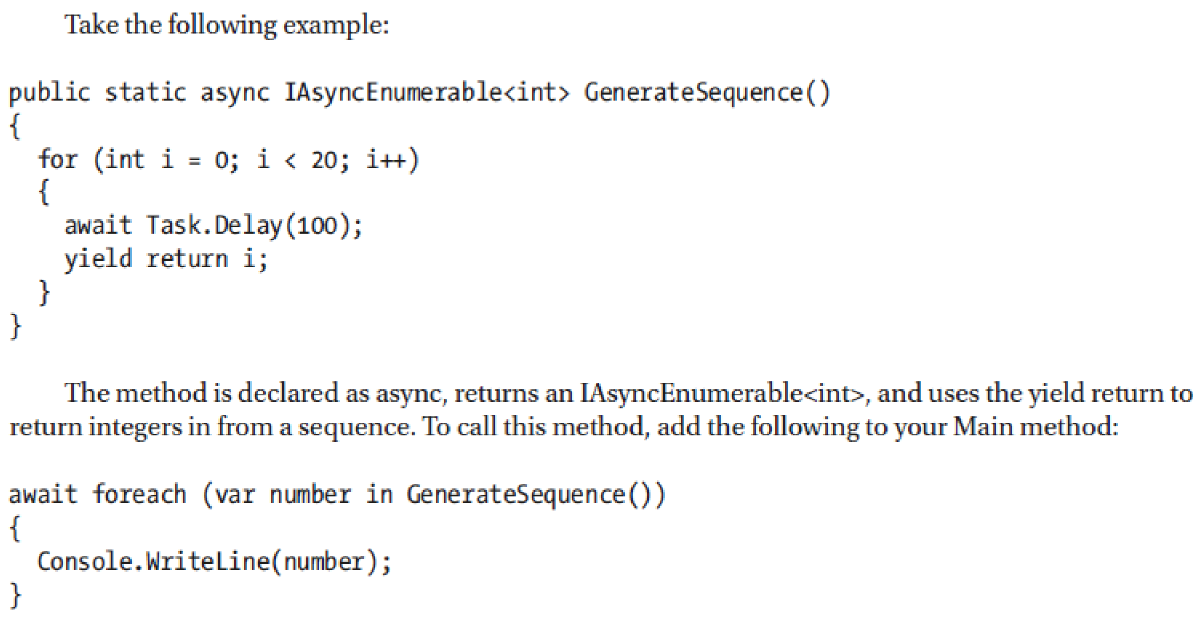


Asynchronous Streams (New 8.0)

New in C# 8.0, streams (covered later in Chapter 20) can be created and consumed asynchronously. A method that returns an asynchronous stream

- Is declared with the async modifier
- Returns an IAsyncEnumerable<T>
- Contains yield return statements (covered in Chapter 8) to return successive elements in the asynchronous stream

Example:



Take the following example:

```
public static async IAsyncEnumerable<int> GenerateSequence()
{
  for (int i = 0; i < 20; i++)
  {
    await Task.Delay(100);
    yield return i;
  }
}
```

The method is declared as async, returns an IAsyncEnumerable<int>, and uses the yield return to return integers in from a sequence. To call this method, add the following to your Main method:

```
await foreach (var number in GenerateSequence())
{
  Console.WriteLine(number);
}
```

Wrapping Up async and await

This section contained a lot of examples; here are the key points of this section:

- Methods (as well as lambda expressions or anonymous methods) can be marked with the `async` keyword to enable the method to do work in a nonblocking manner.
- Methods (as well as lambda expressions or anonymous methods) marked with the `async` keyword will run synchronously until the `await` keyword is encountered.
- A single `async` method can have multiple `await` contexts.
- When the `await` expression is encountered, the calling thread is suspended until the awaited task is complete. In the meantime, control is returned to the caller of the method.
- The `await` keyword will hide the returned `Task` object from view, appearing to directly return the underlying return value. Methods with no return value simply return `void`.
- Parameter checking and other error handling should be done in the main section of the method, with the actual `async` portion moved to a private function.
- For stack variables, the `ValueTask` is more efficient than the `Task` object, which might cause boxing and unboxing.
- As a naming convention, methods that are to be called asynchronously should be marked with the `Async` suffix.

NuGet and .NET Core

NuGet and .NET Core

NuGet is the package manager for .NET and .NET Core. It is a mechanism to share software in a format that .NET Core applications understand and is the default mechanism for loading .NET Core and its related framework pieces (ASP.NET Core, EF Core, etc.). Many organizations package their standard assemblies for cross-cutting concerns (like logging and error reporting) into NuGet packages for consumption into their line-of-business applications.

Packaging Assemblies with NuGet

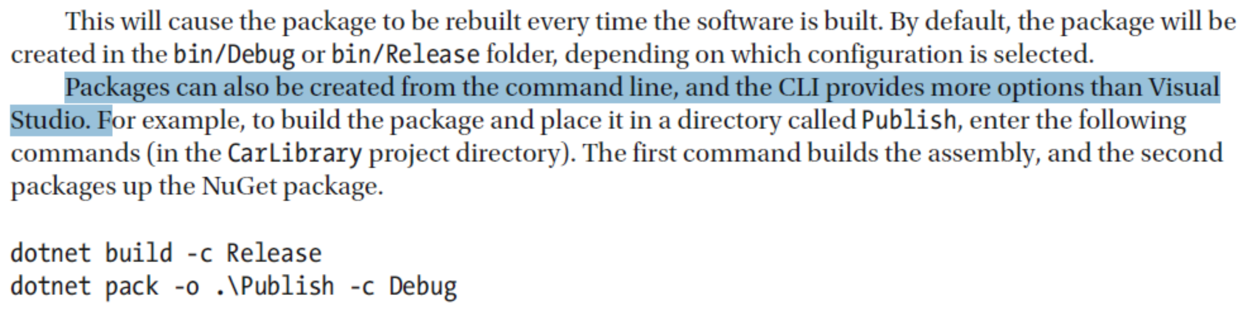
The NuGet Package properties can be accessed from the project's property pages. Right-click the CarLibrary project and select Properties. Navigate to the Package page and see the values that we entered before to customize the assembly. There are additional properties that can be set for the NuGet package (i.e., license agreement acceptance and project information such as URL and repository location).

Note All of the values in the Visual Studio Package page UI can be entered into the project file manually, but you need to know the keywords. It helps to use Visual Studio at least once to fill everything out, and then you can edit the project file by hand. You can also find all of the allowable properties in the .NET Core documentation.

For this example, we don't need to set any additional properties except to check the "Generate NuGet package on build" check box or update the project file with the following:

```
<PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <Copyright>Copyright 2020</Copyright>
    <Authors>Phil Japikse</Authors>
    <Company>Apress</Company>
    <Product>Pro C# 9.0</Product>
    <PackageId>CarLibrary</PackageId>
    <Description>This is an awesome library for cars.</Description>
    <AssemblyVersion>1.0.0.1</AssemblyVersion>
    <FileVersion>1.0.0.2</FileVersion>
    <Version>1.0.0.3</Version>
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
  </PropertyGroup>
```

Command Line can be used:

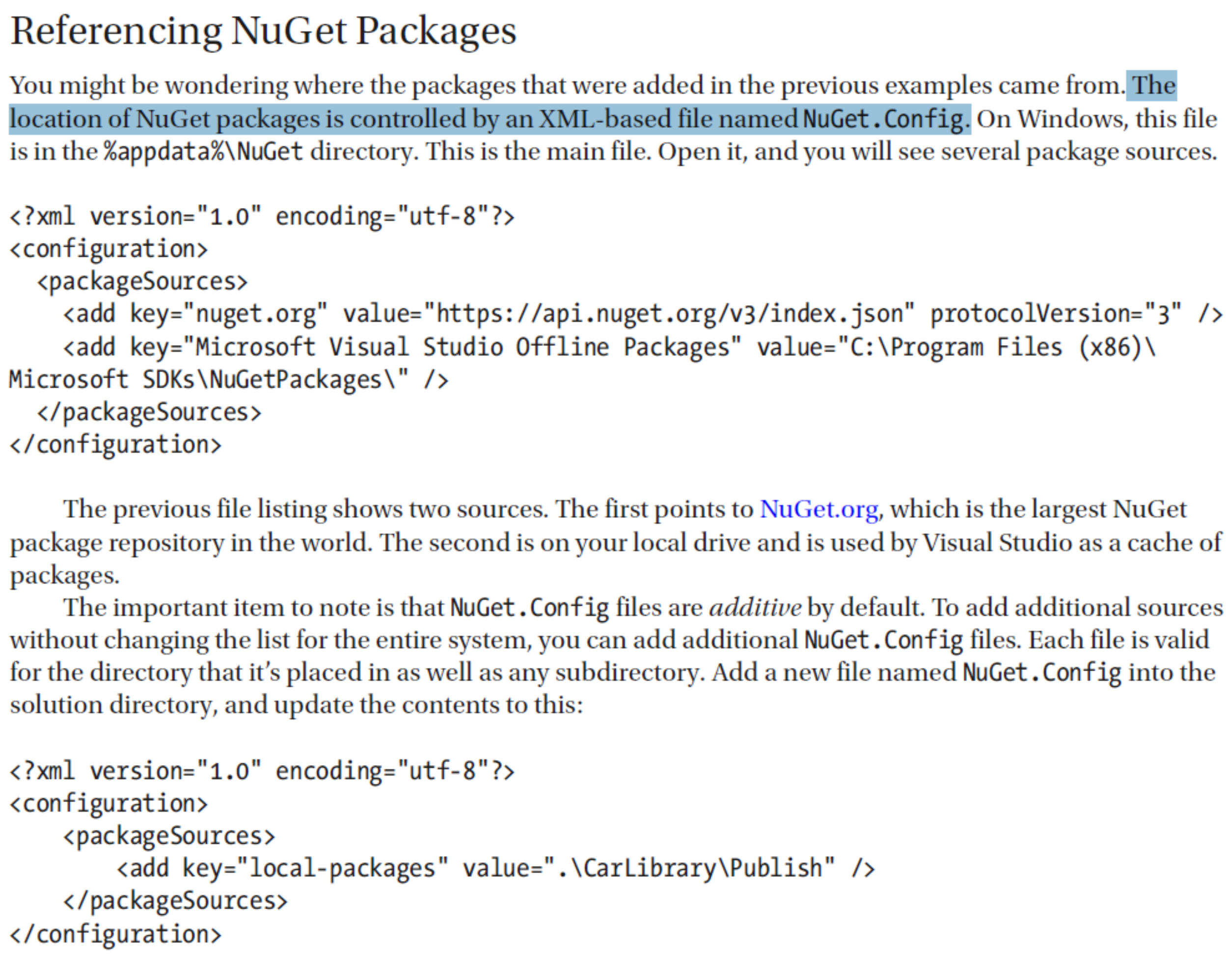


This will cause the package to be rebuilt every time the software is built. By default, the package will be created in the `bin/Debug` or `bin/Release` folder, depending on which configuration is selected.

Packages can also be created from the command line, and the CLI provides more options than Visual Studio. For example, to build the package and place it in a directory called `Publish`, enter the following commands (in the `CarLibrary` project directory). The first command builds the assembly, and the second packages up the NuGet package.

```
dotnet build -c Release
dotnet pack -o .\Publish -c Debug
```

Referencing packages



Referencing NuGet Packages

You might be wondering where the packages that were added in the previous examples came from. The location of NuGet packages is controlled by an XML-based file named `NuGet.Config`. On Windows, this file is in the `%appdata%\NuGet` directory. This is the main file. Open it, and you will see several package sources.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json" protocolVersion="3" />
    <add key="Microsoft Visual Studio Offline Packages" value="C:\Program Files (x86)\
Microsoft SDKs\NuGetPackages\" />
  </packageSources>
</configuration>
```

The previous file listing shows two sources. The first points to NuGet.org, which is the largest NuGet package repository in the world. The second is on your local drive and is used by Visual Studio as a cache of packages.

The important item to note is that `NuGet.Config` files are *additive* by default. To add additional sources without changing the list for the entire system, you can add additional `NuGet.Config` files. Each file is valid for the directory that it's placed in as well as any subdirectory. Add a new file named `NuGet.Config` into the solution directory, and update the contents to this:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <packageSources>
        <add key="local-packages" value=".\CarLibrary\Publish" />
    </packageSources>
</configuration>
```

Reset:

You can also reset the list of packages by adding `<clear/>` into the `<packageSources>` node, like this:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <clear />
    <add key="local-packages" value=".\CarLibrary\Publish" />
    <add key="NuGet" value="https://api.nuget.org/v3/index.json" />
  </packageSources>
</configuration>
```

Note If you are using Visual Studio, you will have to restart the IDE before the updated `nuget.config` settings take effect.

Publishing:

Publishing Console Applications (Updated .NET 5)

Now that you have your C# CarClient application (and its related CarLibrary assembly), how do you get it out to your users? Packaging up your application and its related dependencies is referred to as *publishing*. Publishing .NET Framework applications required the framework to be installed on the target machine, and .NET Core applications can also be published in a similar manner, referred to as a *framework-dependent* deployment. However, .NET Core applications can also be published as a *self-contained* application, which doesn't require .NET Core to be installed at all!

When publishing applications as self-contained, you must specify the target runtime identifier. The runtime identifier is used to package your application for a specific operating system. For a full list of available runtime identifiers, see the .NET Core RID Catalog at `https://docs.microsoft.com/en-us/dotnet/core/rid-catalog`.

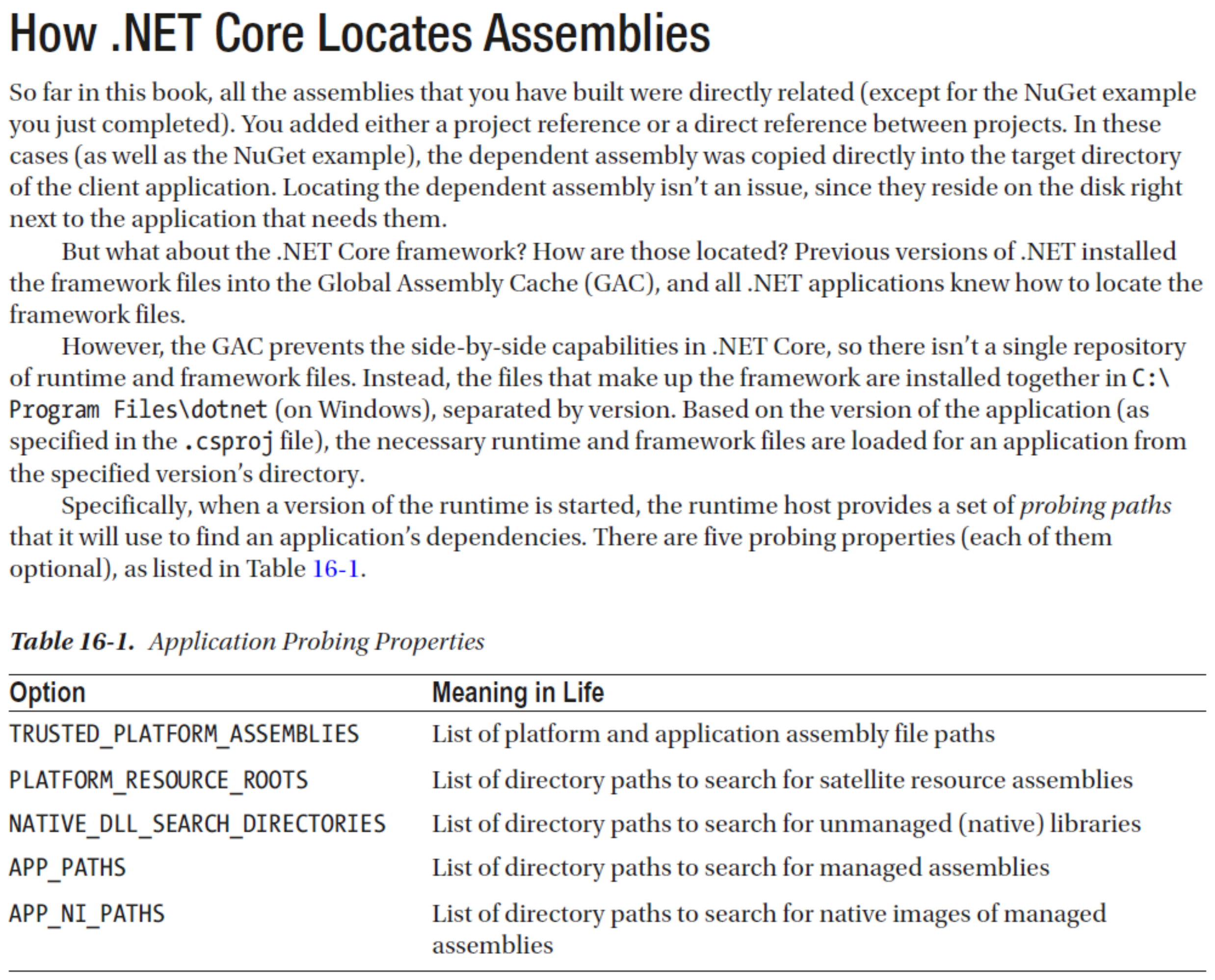
Note Publishing ASP.NET Core applications is a more involved process and will be covered later in this book.

Publishing Framework-Dependent Applications

Framework-dependent deployments is the default mode for the `dotnet publish` command. To package your application and the required files, all you need to execute with the CLI is the following command:

```
dotnet publish
```

Locate Assemblies



How .NET Core Locates Assemblies

So far in this book, all the assemblies that you have built were directly related (except for the NuGet example you just completed). You added either a project reference or a direct reference between projects. In these cases (as well as the NuGet example), the dependent assembly was copied directly into the target directory of the client application. Locating the dependent assembly isn't an issue, since they reside on the disk right next to the application that needs them.

But what about the .NET Core framework? How are those located? Previous versions of .NET installed the framework files into the Global Assembly Cache (GAC), and all .NET applications knew how to locate the framework files.

However, the GAC prevents the side-by-side capabilities in .NET Core, so there isn't a single repository of runtime and framework files. Instead, the files that make up the framework are installed together in `C:\Program Files\dotnet` (on Windows), separated by version. Based on the version of the application (as specified in the `.csproj` file), the necessary runtime and framework files are loaded for an application from the specified version's directory.

Specifically, when a version of the runtime is started, the runtime host provides a set of *probing paths* that it will use to find an application's dependencies. There are five probing properties (each of them optional), as listed in Table 16-1.

Table 16-1. *Application Probing Properties*

Option	Meaning in Life
`TRUSTED_PLATFORM_ASSEMBLIES`	List of platform and application assembly file paths
`PLATFORM_RESOURCE_ROOTS`	List of directory paths to search for satellite resource assemblies
`NATIVE_DLL_SEARCH_DIRECTORIES`	List of directory paths to search for unmanaged (native) libraries
`APP_PATHS`	List of directory paths to search for managed assemblies
`APP_NI_PATHS`	List of directory paths to search for native images of managed assemblies