

50166

Programming the .NET Framework 3.5[®]

Sela College



50166

Programming the .NET Framework 3.5[®]

Sela College

i

© 2009 Sela college All rights reserved.

All other trademarks are the property of their respective owners.

This course material has been prepared by:

Sela Software Labs Ltd.

14-18 Baruch Hirsch St. Bnei Brak 51202 Israel

Tel: 972-3- 6176666 Fax: 972-3- 6176667

Copyright: © Sela Software Labs Ltd.

All Materials contained in this book were prepared by Sela Software Labs Ltd. All rights of this book are reserved solely for Sela Software Labs Ltd. The book is intended for personal, noncommercial use. All materials published in this book are protected by copyright, and owned or controlled by Sela Software Labs Ltd, or the party credited as the provider of the Content. You may not modify, publish, transmit, participate in the transfer or sale of, reproduce, create new works from, distribute, perform, store on any magnetic device, display, or in any way exploit, any of the content in whole or in part. You may not alter or remove any trademark, copyright or other notice from copies of the content. You may not use the material in this book for the purpose of training of any kind, internal or for customers, without beforehand written approval of Sela Software Labs Ltd.

The Use of this book

The material in this book is designed to assist the student during the course. It does not include all of the information that will be referred to during the course and should not be regarded as a replacement for reference manuals.

Limits of Responsibility

Sela Software Labs Ltd invests significant effort in updating this book, however, Sela Software Labs Ltd is not responsible for any errors or material which may not meet specific requirements. The user alone is responsible for decisions based on the information contained in this book.

Protected Trademarks

In this book, protected trademarks appear that are under copyright. All rights to the trademarks in this material are reserved to the authors.

SELA wishes you success in the course!

Table of Contents

Module 1 - Introduction

In This Course.....	3
In This Module	4
.NET Framework Overview	5
What is the .NET Framework?	6
.NET Application Execution	7
.NET Type System Overview	8
Visual Studio and Framework Versions	9
Compatibility.....	10
Visual Studio Multi-Targeting	11
Future Roadmap	12
Summary	13

Module 2 – Memory Management (GC)

In This Chapter	4
.NET as a Managed Environment.....	5
Requirements.....	6
.NET Tracing Garbage Collection.....	7
Managed Heap, Next Object Pointer	8
Object Allocation and the Managed Heap.....	9
Object Allocation and the Managed Heap (contd.)	10
When Does a GC Occur?	11
Mark Phase – Roots	12
Mark Phase	14
Sweep Phase	15
GC and Thread Suspension	16
GC Flavors	17

Choosing the Right Flavor	18
Generations.....	19
Generations: Assumptions.....	20
Allocation and Promotion	21
Generations – Collection.....	22
Generations Illustrated	23
Allocations Are Made.....	24
Generation 0 Fills	25
GC Occurs in Generation 0.....	26
Generation 1 Fills	27
GC Occurs in Generation 1.....	28
Interacting with the GC.....	29
Notification That a GC Occurs.....	30
Latency Mode and Collection Mode.....	32
Weak References	33
Weak References (contd.).....	34
Finalization.....	36
Finalization Internals.....	37
Finalization Illustrated	38
Object is No Longer in Use.....	39
GC Occurs.....	40
Finalizer Thread Wakes Up	41
Finalizer Is Done	42
Another GC Occurs.....	43
Avoid Finalization If Possible.....	45
The Dispose Pattern.....	46
Lab: Weak Timer	47
Summary	48

Module 3 – Streams and File I/O

In This Chapter	3
Input and Output Abstraction.....	4
Streams Direct Hierarchy	5
Decorators and Composites.....	6
File Streams.....	8
Working with File Streams	9
Stream Readers and Writers.....	10
Working with Readers / Writers	11
Binary Readers and Writers	12
Writing Binary Data.....	13
Reading Binary Data.....	14
File and Directory Classes	15
Lab: Word Count	17
Summary	18

Module 4 – Serialization

In This Chapter	3
What's Serialization?	4
Why Serialization?	5
.NET Serialization	6
Serializable User Class.....	7
BinaryFormatter.....	8
Controlling Serialization.....	10
Serialization Callbacks.....	11
Custom Serialization	13
Custom Serialization Example.....	14
Serialization Alternatives	15
Lab: Serialization Framework.....	17
Summary	18

Module 5 – Threading and Asynchronous Programming

In This Chapter	4
Multi-Tasking and Multi-Processing	5
Processes and Threads.....	6
One Program, Multiple Threads	7
Why Threads?	8
Why Not Threads?	9
Scheduling at a Glance.....	10
Amdahl's Law	11
Asynchronous Programming Model	12
APM and Files.....	13
APM and Threads.....	15
BeginInvoke and EndInvoke.....	16
Various Ways to End	17
Various Ways to End (contd.)	18
Maintain State With a Callback.....	19
Maintain State with AsyncState.....	20
BackgroundWorker	22
Lab: APM in the WinForms UI.....	24
Queuing Work for Execution.....	25
ThreadPool.QueueUserWorkItem	26
Manual Threading	28
Thread Class	29
When Does It End?.....	30
Abort vs. Interrupt	31
Inter-Thread Communication	32
Shared Data --> Synchronization	34
Shared Data Races	36
Busy Synchronization	37

Critical Sections.....	38
Monitor and Lock.....	39
Deadlocks.....	41
Deadlocks in Code.....	42
Pulse, PulseAll, Wait, WaitAll.....	44
WaitHandle Synchronization	45
Using Events for Synchronization	46
Parallel Extensions for .NET	48
Lab: Thread-Safe Resource Parallelizing Work	49
Summary	50

Module 6 – Application Domains

In This Chapter	3
System Isolation Boundaries.....	4
Application Domain Isolation.....	5
Why AppDomains?.....	6
Properties of an AppDomain	8
Things to Look Out for	9
Creating an AppDomain.....	10
Retrieving Assemblies	11
Unloading an AppDomain	12
Executing Code in an AppDomain.....	13
Crossing AppDomain Boundaries.....	15
Marshal-by-Value.....	16
Marshal-by-Reference	17
.NET Remoting Overview	19
Lab: Plugin Framework.....	21
Summary	22

Module 7 - Interoperability

In This Chapter	5
Overview of Interoperability.....	6
Platform Invoke.....	7
COM Interoperability.....	8
C++/CLI	9
P/Invoke	10
Behind the Scenes.....	11
Marshaling	13
Standard Mappings.....	14
Character Mapping	15
Marshaling Individual Parameters	16
Calling the Windows APIs	17
Structures and Pointers	19
Combining Unsafe Code.....	20
Marshaling Mutable Strings.....	21
Marshaling Delegates	23
Using Reverse P/Invoke	24
Marshaling Delegates – Caution	25
Generating Signatures	27
Lab: Enumerate Windows.....	28
P/Invoke Summary.....	29
COM Interoperability	30
COM Interoperability - Challenges.....	31
COM Objects From Visual Studio	32
Non-Standard Mappings	33
Manual Customization	34
Primary Interop Assemblies	35
Reflection and IDispatch	36
Lifetime Management.....	37
Error Handling	38

Threading Models	39
.NET Objects as COM Components.....	41
Visual Studio Integration.....	42
C++ Clients.....	43
Alternatives for Registration.....	44
Exposing Interfaces	45
Limitations	46
Lab: Dynamic Dispatch Regex Wrapper.....	48
COM Interoperability Summary.....	49
C++/CLI: The Most Powerful .NET Language.....	50
What Is C++ On The CLR?.....	51
C++ Code Generation.....	52
Basic Class Declaration Syntax	53
More Class Declaration Examples.....	54
Declaring Properties	55
Implementing Properties	56
Using Properties.....	57
Delegates and Events.....	58
Delegates and Events (contd.)	59
Delegates and Events (contd.)	60
Virtual Functions	61
Storage And Pointer Model	64
Pointers and Handles	65
Boxing (Value Types).....	66
Marshaling (Interop)	68
Marshaling Framework.....	69
CLR Types in the Native World.....	70
Native Types In The CLR.....	71
Uniform Destruction/Finalization	73
Practical Interop Scenarios	75
Lab: Native FileSystemWatcher Low-Fragmentation Heap Wrapper.....	76

C++/CLI Summary	78
Interoperability Considerations	79
CLR Hosting.....	80
CLR Hosting From 10,000 ft	81
Summary	82

Module 8 – Advanced Topics

In This Chapter	3
.NET Startup Performance	4
Native Image Generator	5
NGEN Example	6
Dynamically Binding to Delegates.....	7
Dynamically Binding to Events.....	9
Event Registration Tricks.....	11
Invoking Events Asynchronously.....	12
Generics and Reflection	14
Generics at Runtime	16
Object Cloning as Serialization.....	17
Assembly Loading Diagnostics	18
Assembly Load Contexts	19
Troubles Faced with Load Contexts	20
.NET 3.5 SP1 Detailed Error.....	21
Summary	23

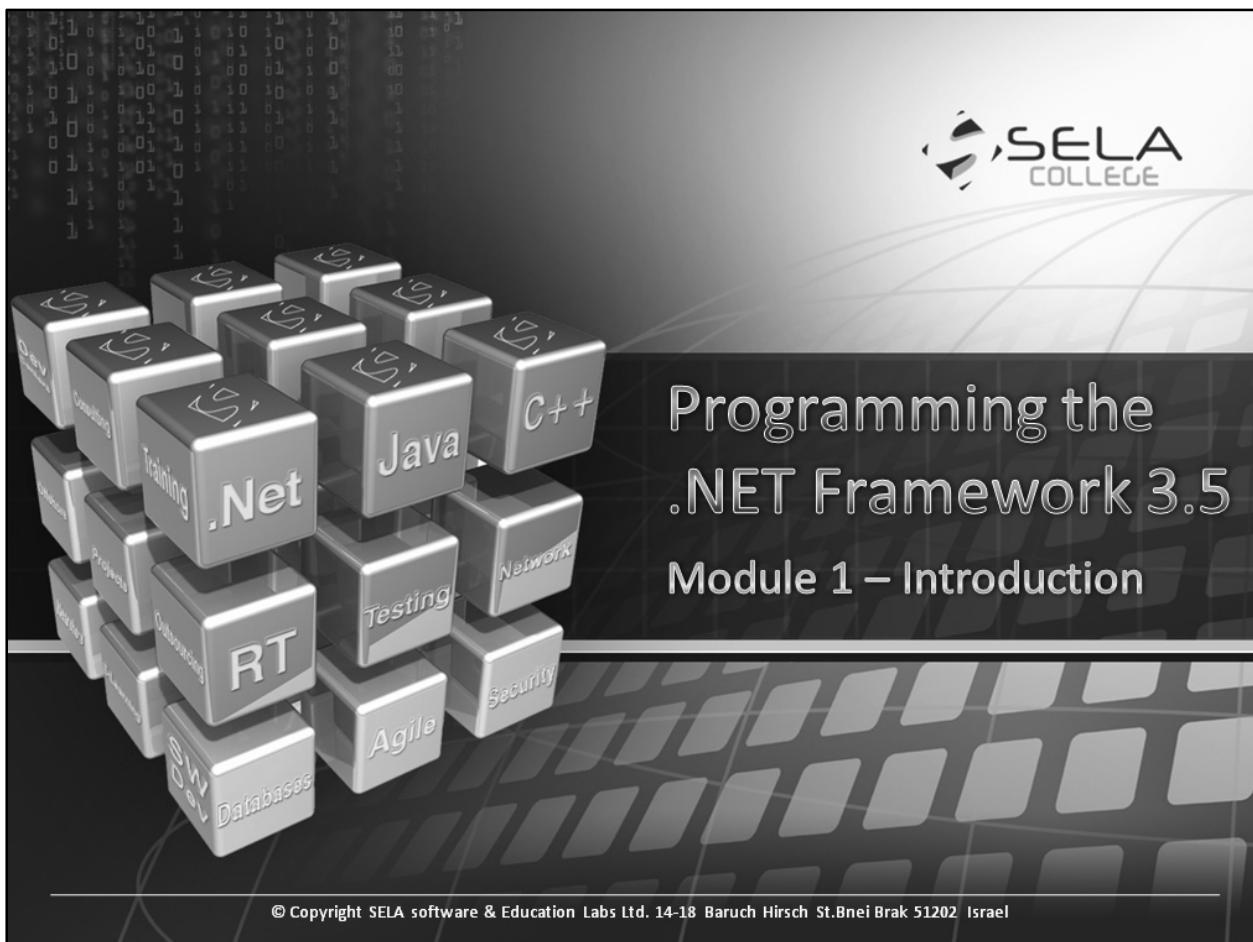
Module 9 - Overviews

In This Chapter	3
ADO.NET.....	4
System.Transactions	5
Windows Communication Foundation	6
Windows Communication Foundation (contd.).....	7
Windows Workflow Foundation	8
Language Integrated Query	9
Related Courses	10
Summary	11

Module 01 - Introduction

Contents:

In This Course	3
In This Module	4
.NET Framework Overview	5
What is the .NET Framework?	6
.NET Application Execution	7
.NET Type System Overview	8
Visual Studio and Framework Versions	9
Compatibility	10
Visual Studio Multi-Targeting	11
Future Roadmap	12
Summary	13



In This Course

- Module 01 – Introduction
- Module 02 – Memory Management
- Module 03 – Streams and File I/O
- Module 04 – Serialization
- Module 05 – Threading and Asynchronous Programming
- Module 06 – Application Domains
- Module 07 – Interoperability
- Module 08 - Advanced Topics
- Module 09– Overviews



1-4 | Module 01 - Introduction

In This Module

- ⌚ Overview of the .NET Framework
- ⌚ Overview of the .NET Type System
- ⌚ Visual Studio and Framework Versions
- ⌚ Compatibility
- ⌚ Visual Studio Multi-Targeting
- ⌚ Future Roadmap



.NET Framework Overview

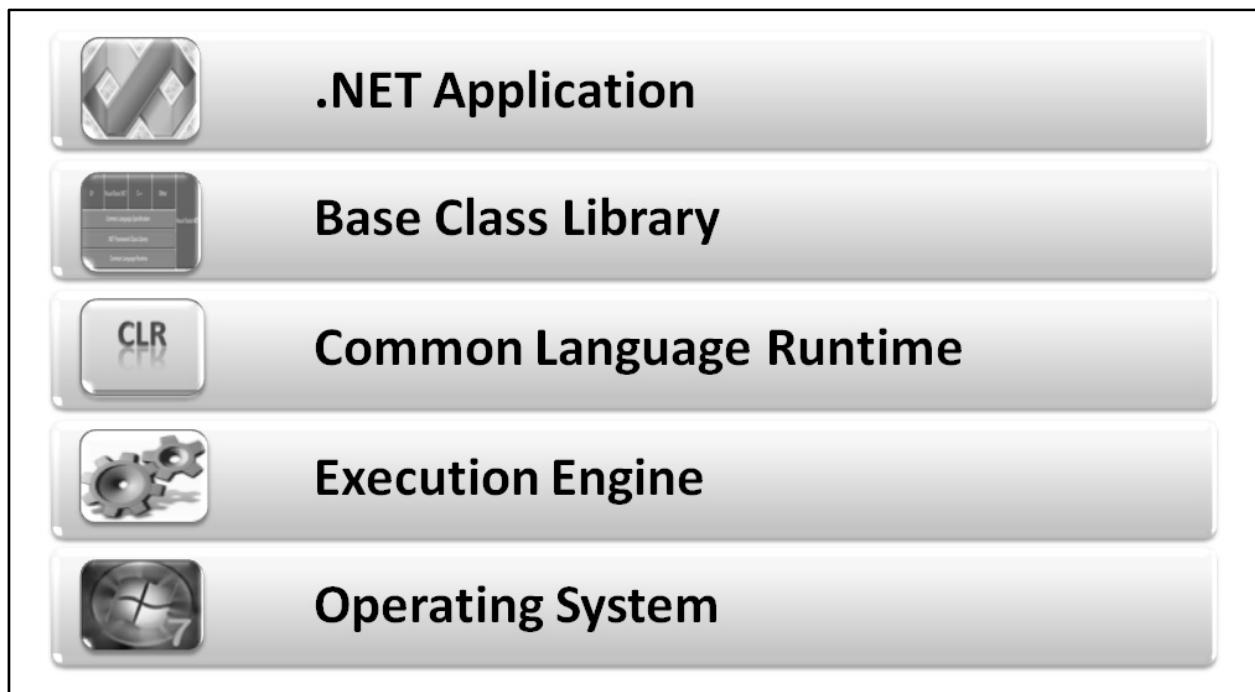


What is the .NET Framework?

Brainstorm with the participants about what really is the .NET Framework.

The .NET Framework is a general name for a variety of things, including the CLR, the execution engine, the FCL, the JIT and many other things. Before we learn about the .NET Framework, it's necessary to set clear expectations about what the .NET Framework is.

What is the .NET Framework?



The .NET Framework is a hierarchy of abstractions that lies on the top of the operating system. The first layer of abstraction is the execution engine, which abstracts away the underlying OS and provides a platform adaptation layer.

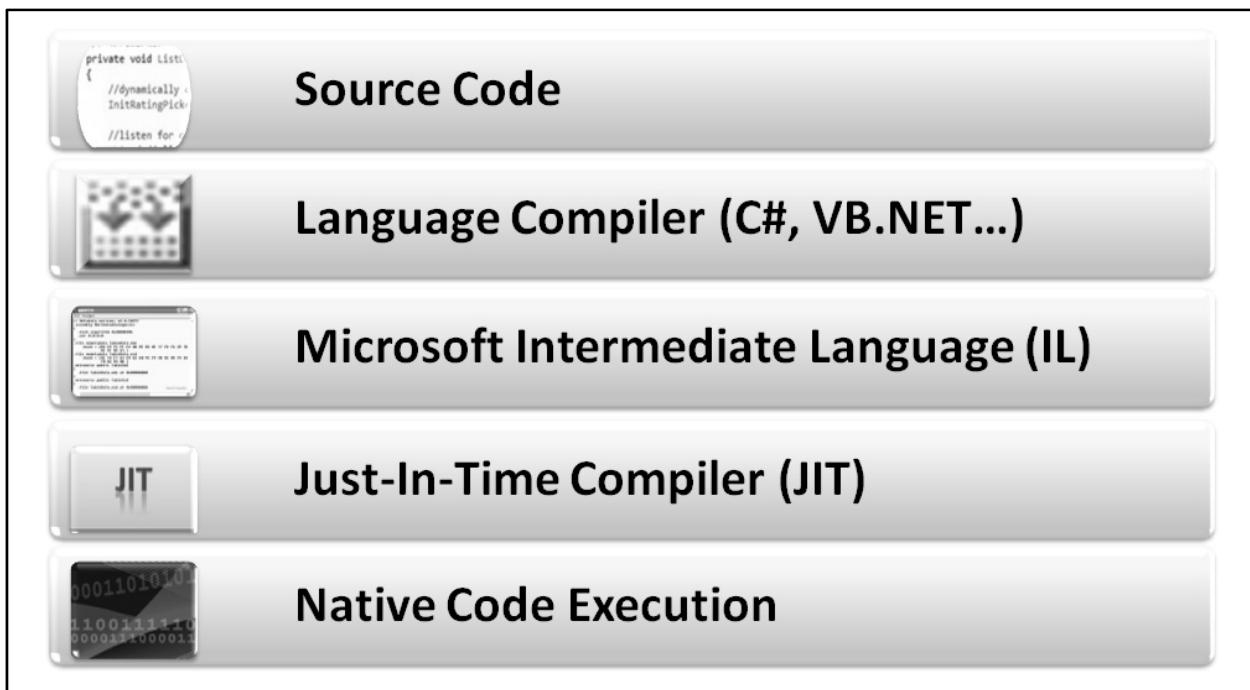
Above that is the CLR, which provides facilities such as assembly and type loading, memory management, metadata and type information, just-in-time compilation etc.

Next is the BCL, which is an enormous collection of well-organized types, also acting as a wrapper for underlying OS functionality.

Finally, our .NET application interacts with the BCL and runs on top of the entire abstraction stack.

In this course we will focus on the BCL and the CLR. Most of our time will be dedicated to features that the application can interact with directly through the BCL.

.NET Application Execution



A .NET application does not execute directly from its source code – managed languages are not interpreted.

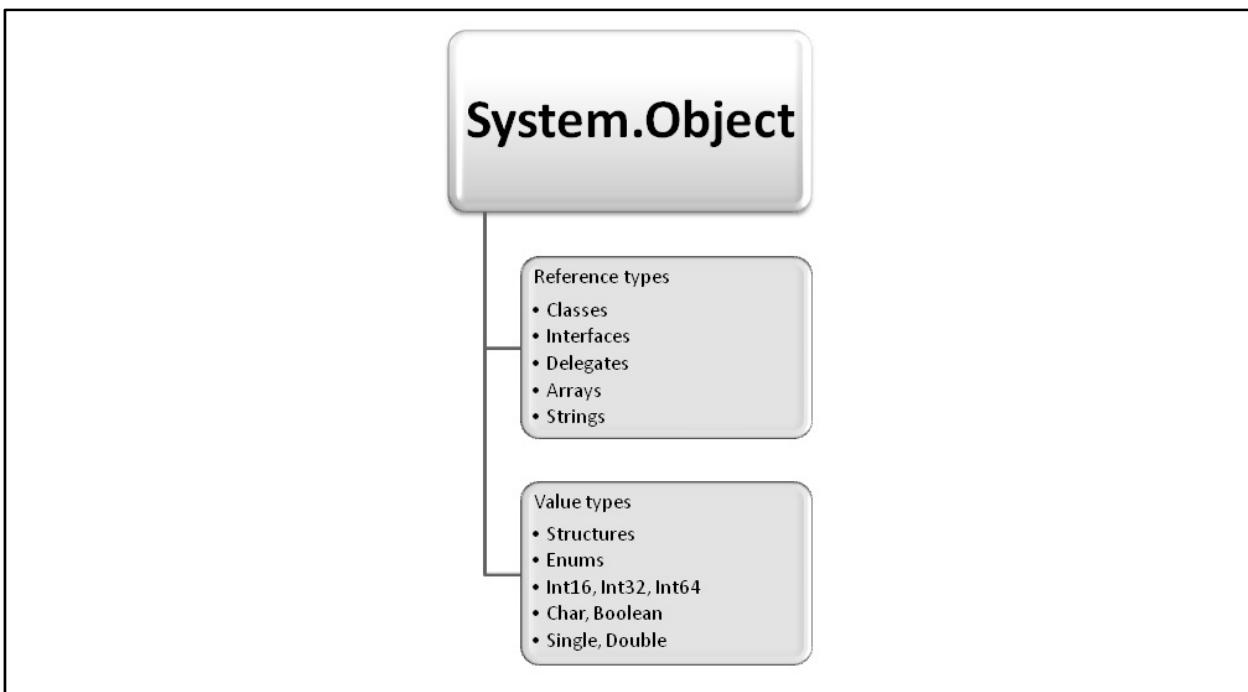
For each .NET language there is a language compiler which compiles the source code to the Microsoft Intermediate Language (IL), which is a low-level language abstracting away the operating system, the bit sizes (32-bit or 64-bit) and other aspects. IL is object-oriented, so type information, virtual methods and other OO-features are preserved.

Next, when the application is executed, a Just-In-Time compiler kicks in to compile the IL code to native machine code that can be executed on the target processor.

This gives the .NET Framework a certain degree of location independence – the same .NET application can execute on a 32-bit system, a 64-bit system, on Windows, on Linux and so on. All that matters is that the operating system has an implementation of the .NET Framework, the CLR and the execution engine required.

MCT USE ONLY. STUDENT USE PROHIBITED

.NET Type System Overview



All .NET types derive from System.Object explicitly or implicitly.

However, there is a clear categorization of types: Value types and Reference types.

Reference types include classes, interfaces, delegates, arrays and strings, and have the following characteristics:

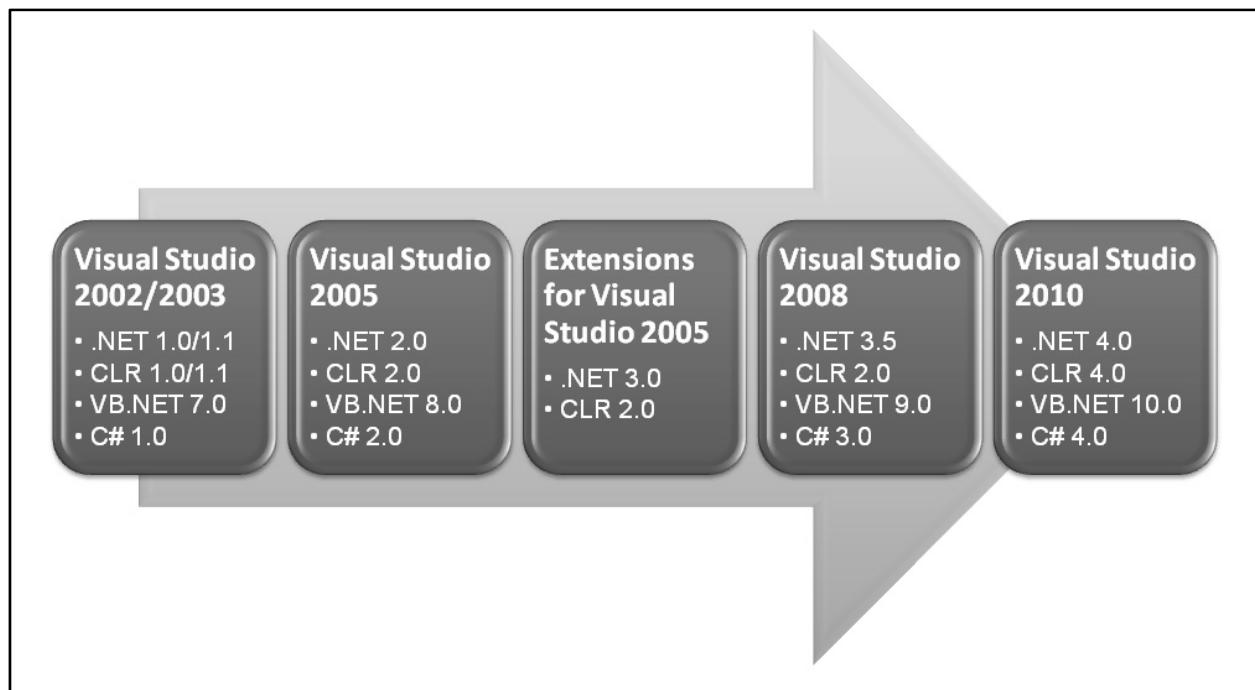
- Accessed by reference
- Copied by reference (when passing as parameters, assigning or returning from a method)
- Compared by reference
- Allocated on the garbage-collected heap (with a minor exception for unsafe stack-allocated arrays)

Value types include structures, enums and the primitive types like int, char, float and bool, and have the following characteristics:

- Accessed by value
- Copied by value
- Compared by value (by the default ValueType.Equals and ValueType.GetHashCode implementations)
- Allocated on the stack (with an exception for boxed value types, which copy a value type to the heap)

The worlds of value types and reference types are bridged by the concept of boxing, which is a process that takes a value type instance and “wraps it in a box”, producing an object that is placed on the garbage-collected heap and can be passed around by reference. This box can be opened by an unboxing process, which converts it back to a value type instance.

Visual Studio and Framework Versions



Visual Studio versions usually dominate the versions of the .NET framework, the main managed languages (VB.NET and C#) and the CLR, but this was not always the case.

For example, the .NET 3.0 release was not accompanied by a CLR change, and was supported only by a set of external extensions installed on top of Visual Studio 2005.

There are also servicing releases of the .NET framework and of Visual Studio, such as .NET 3.5 SP1 and Visual Studio 2008 SP1, which also affect framework versioning and often introduce new features. (For example, .NET 3.5 SP1 introduces Data Services, Entity Framework and other technologies.)

Visual Studio 2010 is the next step on the product roadmap (will be discussed at the end of this module), and it will feature a new version of the .NET framework, a new version of the VB.NET and C# languages and a new version of the CLR.

It's also worth noting that aside from the desktop .NET and CLR, there are also additional editions which feature a subset of the full CLR functionality. For example, the Silverlight CLR features a minimal 4MB installation footprint with an implementation of the CLR that runs within the browser on Windows and Mac. The .NET Compact Framework is another subset that runs on Windows Mobile devices.

Visual Studio is packaged in various forms and has various editions, including the free Visual Studio Express edition, the Professional edition, the Team Suite edition and other flavors. For the purpose of this course, the Visual Studio 2008 Professional edition (with Service Pack 1) or higher is recommended.

Compatibility

- ➊ The BCL is *usually* fully backwards-compatible
- ➋ The CLR is *not always* backwards-compatible

- CLR 4.0 will allow side-by-side Execution.

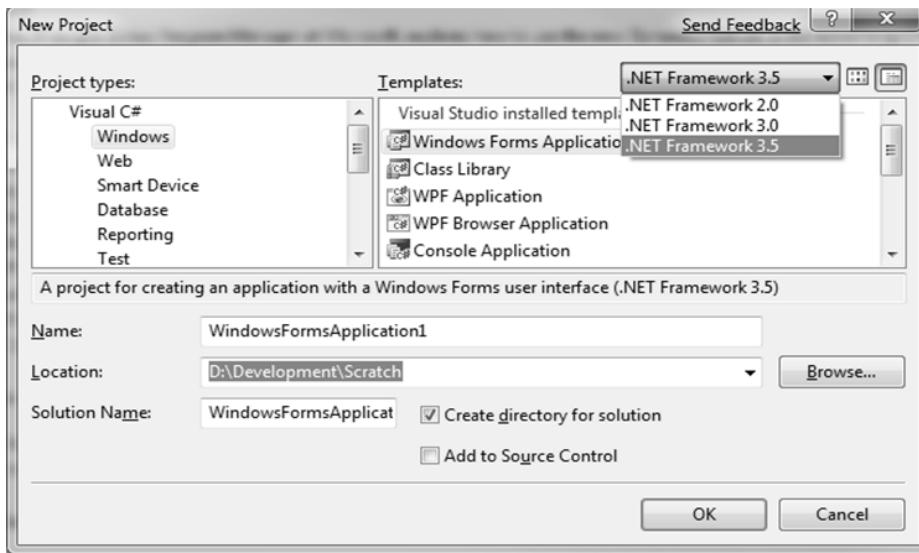
BCL types are usually not broken in framework releases, unless there is a significant security vulnerability of API issue that is encountered. Most of the time, new and replacement APIs are introduced with a different type name (e.g. `TimeZone` and `TimeZoneInfo`). When an API is marked for deprecation, it would usually linger around for one or two releases marked with the `[Obsolete]` compiler attribute, which produces a warning (and eventually will also produce an error) when the API is used.

The CLR on the other hand often involves significant implementation changes with regard to the metadata, GC, JIT and other mechanisms which can't be easily made backwards-compatible. As a result, CLR versions are often incompatible.

CLR 4.0 will be the first CLR version to offer side-by-side execution of CLR 4.0 and CLR 2.0 code in the same OS process. This will guarantee full compatibility for older applications.

Visual Studio Multi-Targeting

- Specify the framework version to target



Previous Visual Studio versions were tied to one version of the .NET framework. For example, Visual Studio 2005 can only target .NET 2.0.

Visual Studio 2008 changes that by giving the programmer a choice of the .NET framework version to target (currently choosing between 2.0, 3.0 and 3.5).

Visual Studio assists the programmer by not allowing to add project reference assemblies which require a higher version of the framework, and enforcing a certain degree of strictness in the managed languages suitable for the framework version.

Future Roadmap

- Visual Studio 2010 CTP
- What's in .NET 4.0?
 - Parallel Extensions
 - Oslo, Dublin, M, WCF 4.0, WF 4.0
 - WPF 4.0
 - ...and many other technologies

Summary

- Overview of the .NET Framework
- Overview of the .NET Type System
- Visual Studio and Framework Versions
- Compatibility
- Visual Studio Multi-Targeting
- Future Roadmap





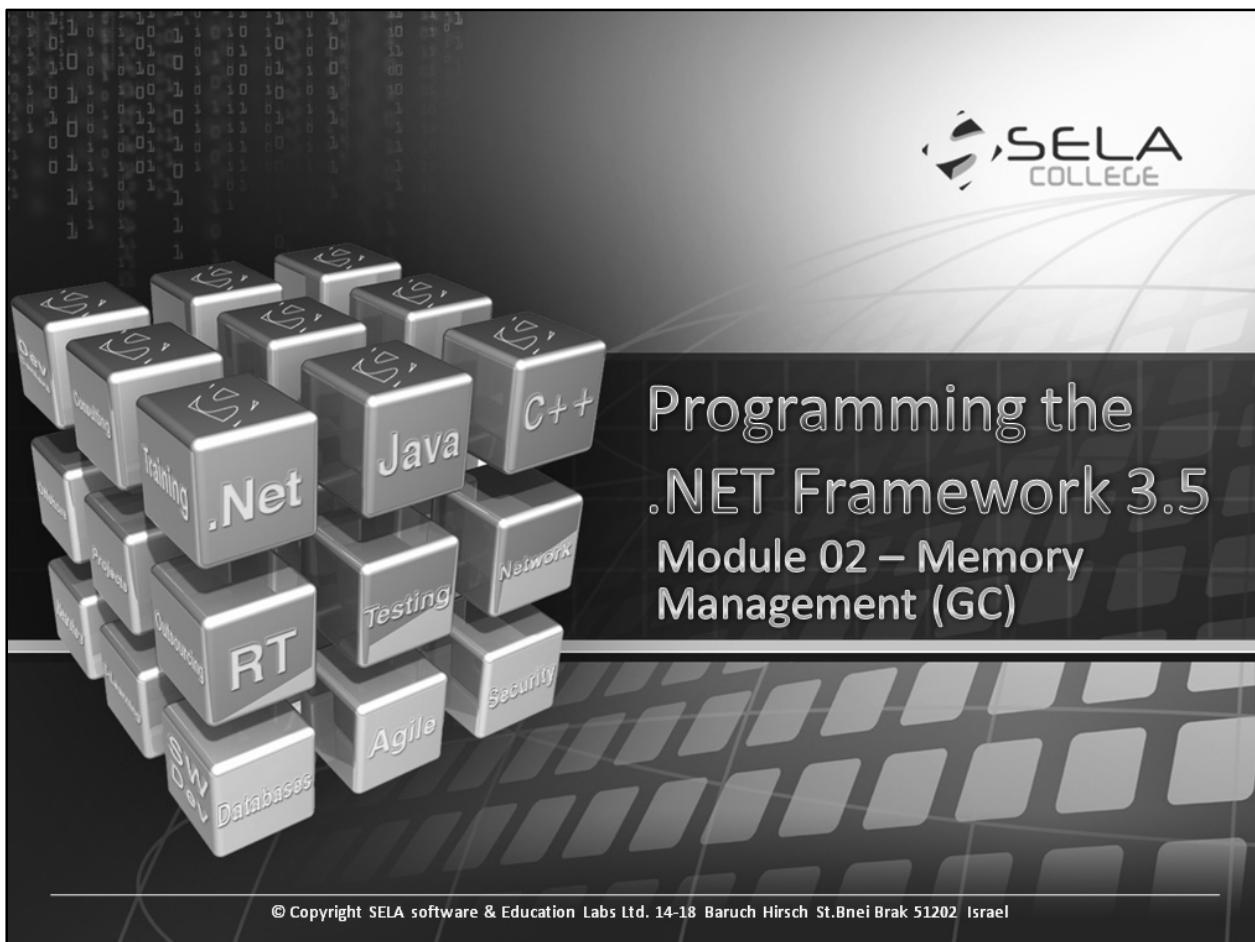
Module 02 - Memory Management (GC)

Contents:

In This Chapter.....	4
.NET as a Managed Environment.....	5
Requirements.....	6
.NET Tracing Garbage Collection	7
Managed Heap, Next Object Pointer	8
Object Allocation and the Managed Heap	9
Object Allocation and the Managed Heap (contd.).....	10
When Does a GC Occur?	11
Mark Phase – Roots.....	12
Mark Phase	14
Sweep Phase	15
GC and Thread Suspension.....	16
GC Flavors	17
Choosing the Right Flavor.....	18
Generations.....	19
Generations: Assumptions	20
Allocation and Promotion	21
Generations – Collection.....	22
Generations Illustrated	23
Allocations Are Made	24
Generation 0 Fills	25
GC Occurs in Generation 0	26
Generation 1 Fills	27
GC Occurs in Generation 1	28

MCT USE ONLY. STUDENT USE PROHIBITED

Interacting with the GC.....	29
Notification That a GC Occurs	30
Latency Mode and Collection Mode.....	32
Weak References.....	33
Weak References (contd.)	34
Finalization	36
Finalization Internals	37
Finalization Illustrated	38
Object is No Longer in Use	39
GC Occurs.....	40
Finalizer Thread Wakes Up.....	41
Finalizer Is Done	42
Another GC Occurs.....	43
Avoid Finalization If Possible	45
The Dispose Pattern	46
Lab: Weak Timer.....	47
Summary	48



In This Chapter

- ⌚ Overview of memory management
- ⌚ Garbage collection first steps
- ⌚ GC flavors
- ⌚ Generations
- ⌚ Interacting with the GC
- ⌚ Weak references
- ⌚ Finalization and Dispose
- ⌚ Lab



.NET as a Managed Environment

- .NET is a managed environment
- Memory management is difficult
- The *garbage collector* (GC) takes care of memory management

Given that .NET is a managed environment, the CLR and execution engine can perform memory management tasks that are impossible in a purely native application. Explicit memory management is usually associated with quite a few problems, including forgetting to free memory (memory leaks), trying to access memory that was already freed, freeing memory more than once (double-delete) and others.

The garbage collector takes care of memory management by implicitly managing the lifetime of all objects, and freeing the memory associated with these objects when it is no longer required. Developers no longer have to deal explicitly with memory management operations, at least for purely managed resources. (Later in this chapter we will also examine unmanaged resources.)

Requirements

- GC design goals
- First GC implementation – Lisp, circa 1963

Brainstorm the design goals / requirements from a garbage collector.

Examples of requirements:

- Configurability and traceability
- Correctness (obviously...)
- Execution of cleanup code (finalization)
- Performance
- ... and many others!

GC is not a new idea – the first implementation dates back to Lisp. It was a copying garbage collector, meaning that memory was divided into two separate spaces (one of them backed by tape storage) and when memory would run out in one of the spaces, all living objects would be copied to the second space and the first space would be reclaimed. This process would repeat, cycling between the storage spaces when a garbage collection occurs. This is clearly not a very relevant implementation, primarily because of its performance penalty.

An alternative implementation that is somewhat in use today is a *reference counting garbage collector*, which maintains a reference count for each object and deletes objects when they are no longer referenced. This scheme is used by COM to manage the lifetime of COM objects and by the Windows kernel to manage the lifetime of kernel objects (through handles and direct kernel references). It is also associated with a performance cost, and does not alleviate the problem of cycles, so we will not focus on it today.

.NET Tracing Garbage Collection

- Objects are collected at non-deterministic times
- No promise of deterministic finalization
- No overhead while the GC is idle

The .NET garbage collector is a *tracing garbage collector*. It reclaims unused memory at non-deterministic times. The biggest promise of the .NET GC is that it does not incur (almost) any overhead while the application is running. Only when the garbage collector kicks in and constructs a graph of all reachable objects, later reclaiming the unused memory – the performance penalty is paid.

On the other hand, it's impossible to guarantee the deterministic execution of cleanup code. Deterministic execution of cleanup code is something that the application must take care of explicitly, and is discussed later in this chapter (under "Finalization and Dispose").

Managed Heap, Next Object Pointer

- A managed heap is created on initialization
- A pointer points to the next object
- An object allocation increments the pointer and returns the previous address

Every managed application is associated with at least one managed heap when it is initialized. This heap is not static and can grow as needed by requesting more memory from the operating system. Additionally, a pointer (called *next object pointer*) is maintained, always pointing to the next available location where an object can be allocated.

When an allocation is triggered by operator *new*, the next object pointer is incremented by the size of the allocated object, and its previous value is returned.

This guarantees an extremely low cost for object allocations, and as long as the heap space is not exhausted (which requires a GC), a heap allocation is comparable in cost to a stack allocation, meaning it is extremely cheap.

In contrast to the much more expensive C-runtime heap, where an object allocation requires walking through a linked-list to find a big enough memory block, splitting it and updating all relevant pointers to keep the list valid – this model means that every allocation consists only of incrementing a pointer.

Moreover, on most heaps (like the C-runtime one), objects are allocated wherever free space found. This might generate a situation where consecutively created objects might be separated in the address space, thus harming CPU cache locality. The .NET heap ensures that such objects will be allocated contiguously in memory.

In C, a call to **malloc()** typically results in a search of a linked list of free blocks. This can be time consuming, especially if your heap is badly fragmented. To make matters worse, several implementations of the C run time lock the heap during this procedure. Once the memory is allocated or used, the list has to be updated. In a garbage-collected environment, allocation is free, and the memory is released during collection.

Object Allocation and the Managed Heap

```
Employee e = new Employee();
e = NOP;
NOP = NOP + sizeof(Employee);
```

New Object Pointer

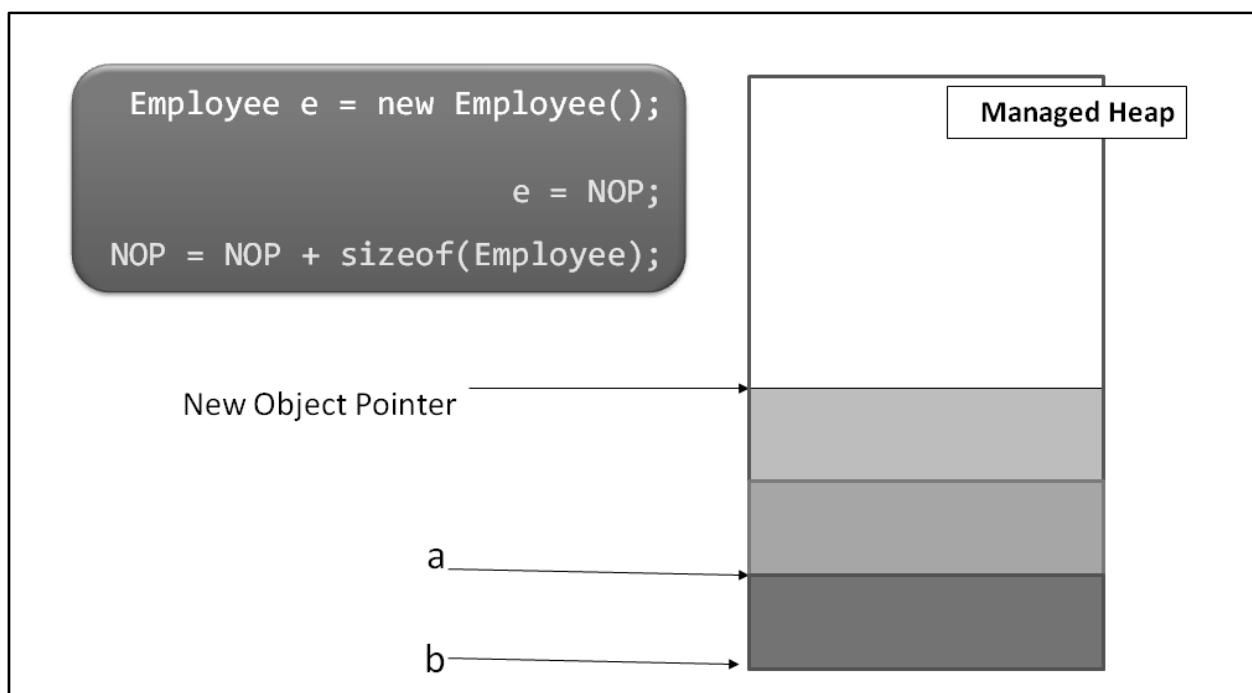
a

b

Managed Heap

This diagram demonstrates the managed heap before an object allocation. The allocation consists of returning the previous value of the next object pointer and incrementing it by the size of the allocated object.

Object Allocation and the Managed Heap (contd.)



This diagram demonstrates the managed heap after an object allocation. The allocation consists of returning the previous value of the next object pointer and incrementing it by the size of the allocated object.

When Does a GC Occur?

- In *this* model, a GC occurs when memory runs out
- The GC has to free memory (**Sweep**) by detecting unreferenced objects (**Mark**)



How does it detect that an object is unreferenced?

At some point, nonetheless, memory will run out. In this model, for simplicity's sake, we will assume that a garbage collection occurs only when managed heap memory completely runs out. The garbage collector is then invoked to free some memory by detecting unreferenced objects and reclaiming the memory they occupy. These phases are called Mark and Sweep: during the mark phase, the GC determines which objects need be kept alive, and during the sweep phase it reclaims the memory occupied by the unreferenced objects (which are dead as far as the application is concerned).

How can the garbage collector detect whether an object is referenced or not? The algorithm for determining this is the reason the .NET GC is called a *tracing* garbage collector – it has to trace through all object references to construct a full graph of all live objects.

2-12 | Module 02 - Memory Management (GC)

Mark Phase – Roots

- An application has a set of **roots**
 - **Static** object references
 - **Local** objects in currently active methods
 - Other types (GC handles, finalization queue, ...)
- ?
- Where does **objRef** stop being an active root?

In Release Mode

In Debug Mode

```
static public void Main() {  
    object objRef = ...;  
    int i = objRef.GetHashCode();  
    PerformLengthyCalculation(i);  
}
```

During the mark phase, the garbage collector strives to construct a full graph of the application's reachable objects (the rest of the objects will be deemed unreachable as a corollary).

Constructing a graph requires a set of starting points, which are called *roots*. There are various types of roots, but the one thing that is common to all of them is that they are not reachable or referenced on their own – they form the starting points of the graph. (This does not mean, however, that another object can't have a reference to a root; it only means that the roots do not *have* to be referenced in order to be considered roots.)

Examples of roots include static object references, like a static reference field within a class; local objects on the stacks of currently active methods; and other types of roots which we will discuss later.

Local roots are particularly tricky because it's entirely possible for a local variable to be considered unreferenced before it actually goes out of its method's scope. For example, consider the code on the slide. Where does the *objRef* local reference stop being an active root – at the end of the method, or at the line where it last used?

Local Roots:
System.Threading.Timer



See the **LocalRoots_Timer** project in the **Module02_MemoryManagement_GC** solution under the SampleCode folder for more information about this demo in the code comments.

This demo shows that a local root is no longer considered a root after the execution passed the last instruction where it was actually used by its containing method. (This is the Release build behavior – in Debug builds, the GC retains the roots until the end of the scope to allow their inspection within a debugger.)

This is implemented by assistance from the just-in-time compiler, which constructs reachability tables for local variables, and is outside the scope of this course.

Mark Phase

- The GC builds a graph of all reachable objects
- The rest is considered “garbage”
- To extend an object’s lifetime, use the **GC.KeepAlive** method

During the mark phase, the garbage collector recursively traverses the set of active roots to construct a graph of all reachable objects. This graph may contain cycles, and therefore the GC also marks objects that it visits so that it does not enter an infinite loop. Eventually, the reachable objects are all known to the garbage collector, and the rest of the objects are known to be unreachable – there is no way for the application to reach and use these objects. Therefore, their memory can be reclaimed.

To extend the lifetime of an object referenced by a local reference, for example if it is being held by unmanaged code, use the `GC.KeepAlive` method and pass to it as a parameter the object reference. The `KeepAlive` method does not do anything in particular – it’s simply the fact that the object reference is passed as a parameter to a non-inlined method that causes the GC to retain the object alive at least until the instruction that calls `GC.KeepAlive`.

Sweep Phase

- The GC compacts the heap by moving live objects close together
- The GC updates the next object pointer

During the sweep phase, the GC has a graph of all reachable objects in the applications, and as a corollary, the information about memory regions that are not occupied by any live objects. To maintain the next object pointer abstraction (which makes memory allocation blazingly fast), the GC shifts all live objects close together, compacting the holes occupied by dead objects. This is essentially a de-fragmentation process.

During the sweep phase, object references are updated to point to their object's new location. This means that .NET references are not stable pointers – they can move around when a garbage collection occurs. This is the primary reason why pinning is required when passing the address of a managed object to unmanaged code. (Pinning is outside the scope of this chapter.) After the sweep, the next object pointer is updated to point to the beginning of the freed memory region, meaning that future allocations can be satisfied without searching the managed heap for an available “hole” where objects can be allocated. (Unlike the CRT allocator, for example, as discussed in previous slides.)

GC and Thread Suspension

- During GC, all managed threads must be suspended
- Threads running unmanaged code can still execute

During the garbage collection process, threads must be suspended so that they do not modify the GC graph. Consider a GC process that deems an object alive but in the meantime another thread removes the last reference to that object, rendering it unreachable. Another example is a GC process which has begun compacting the managed heap, but in the meantime threads are accessing the objects on the heap using old references, which corrupts the managed heap.

It appears that any managed code running during a garbage collection *must* be suspended so that it does not interfere with the GC's work. We will look into GC flavors to see that it is not always the case, and that it *is* possible for managed code to execute simultaneously with at least the Mark phase of the garbage collection process.

Note that unmanaged code does not bother us at all, because it is guaranteed that unmanaged code cannot access object references directly and affect the GC's work. Again, pinning prevents objects that *were* passed to managed code from moving around during GC, guaranteeing the integrity of the process.

GC Flavors

- GC Flavors optimize GC for specific application types
- Client GC (alias **Workstation GC**) is optimized for low latency
- **Server GC** is optimized for high throughput and scalability

Not all applications require the same characteristics from the .NET garbage collector. For example, a client application favors a low-latency GC, which takes a minimal amount of time to complete so that the application's UI can be unfrozen. On the other hand, a server applications favors a highly scalable GC which minimizes contention and utilizes all available processors to reclaim unused memory.

There are three GC flavors (with a fourth being added in CLR 4.0, and not discussed here):

- Workstation concurrent GC
- Workstation non-concurrent GC
- Server GC

The workstation non-concurrent GC flavor has the following characteristics:

- GC occurs on the thread triggering the GC
- All threads are safely paused during the Mark and Sweep phases
- Execution resumes when GC is complete

The workstation concurrent GC flavor has the following characteristics:

- GC occurs on a dedicated GC thread that is created when needed and destroyed after a timeout period of inactivity
- Threads are not paused during the Mark phase – the GC alleviates the possible race conditions by carefully constructing the graph and receiving cooperation from the allocator's implementation
- All threads are safely paused during the Sweep phase
- Execution resumes when GC is complete

The server GC flavor has the following characteristics:

- There is a separate managed heap section for each CPU available to the process
- Allocations performed by a thread running on a specific CPU are satisfied from the managed heap section associated with that CPU
- During the Mark and Sweep phases, all threads are safely paused
- GC happens in parallel on dedicated GC threads, one for each processor, which concurrently perform the GC on the managed heap sections associated with each CPU
- Execution resumes when GC is complete

Choosing the Right Flavor

- Choosing between the two models is usually automatic
- Can be controlled through app.config

What are the pros and cons?



The default CLR host which runs console applications, UI applications, services and other types of managed code which do not execute in a hosting environment defaults to the workstation concurrent GC flavor, even when executing on a multi-processor server machine.

Specific hosting environments have the ability to control the GC flavor. For example, ASP.NET on a multi-processor machine uses the server GC automatically. You can also control it by writing a CLR host or by using an application configuration file with the `<gcConcurrent>` and `<gcServer>` elements. The only limitation is that a single-processor machine cannot take advantage of server GC, and will automatically revert to workstation GC.

What are the pros and cons of each flavor? What are characteristic scenarios?

Here are a few examples:

- The UI thread never triggers collections: WKS
- The UI thread triggers collections: WKS NC
- Server app with lots of allocations: SVR

Generations

- A full GC is extremely expensive!
 - Linear in # of referenced objects
- The heap is divided into generations
 - This enables a partial collection process

The GC algorithm described in the previous slides is extremely expensive. When heap memory is exhausted, a full mark phase and full sweep phase occurs, touching every live object and moving around huge chunks of data. In modern applications, which often use hundreds of megabytes and even gigabytes of memory, it is not practical to perform a GC only when memory is exhausted, for performance reasons.

The real-world optimization that enables a partial collection process is called the *generational GC model*. By partitioning the managed heap into sections called *generations*, a partial collection can occur in certain scenarios, alleviating the need for a full mark-and-sweep which are extremely expensive.

Generations: Assumptions

- Collecting a portion of the heap is faster than collecting the whole heap
- The newer an object is, the shorter will be its lifetime
- The older an object is, the longer will be its lifetime

These are the assumptions that drive the generational model. The first one is obvious, but the other two are not quite as intuitive. However, on the large scale, statistically, these assumptions make sense – almost every application has a set of always-alive objects which are created close to the application’s startup and destroyed when the application shuts down, and a large amount of temporary, short-lived objects which are created as a result of some outside interaction (e.g. a user request to a web server) and destroyed when the interaction completes.

Nonetheless, these are just assumptions – an application that behaves differently from these assumptions will not benefit from the generational model. One of the primary ways to optimize memory management performance in a managed application is to make sure short-lived objects die quickly and long-lived objects linger on for as long as possible. (This is the opposite of a phenomenon termed *mid-life crisis*.)

Allocation and Promotion

- New objects are allocated in **generation 0**
- When generation 0 fills, a GC occurs in **generation 0**
- Survivors are promoted to **generation 1**
- And so on – up to **generation 2**

Objects are always allocated in generation 0 (except for large objects – discussed later). Generation 0 is a small part of the managed heap, typically around 128KB – 4MB in size (depending on process bitness and other factors). Generation 1 is slightly larger, and generation 2 is the rest of the managed heap – so it can grow to the maximum provided by the host operating system.

When a generation is filled, a garbage collection *within that generation* occurs. Objects that survive are promoted to the next generation if there is one. Certain factors might affect the promotion ability, such as pinning and other run-time considerations, but these are outside the scope of this chapter.

The last generation (generation 2 in the .NET GC model) is collected not only when it becomes full, but also periodically when allocation watermarks are reached. These watermarks are dynamically assigned and are outside the scope of this chapter.

Generations – Collection

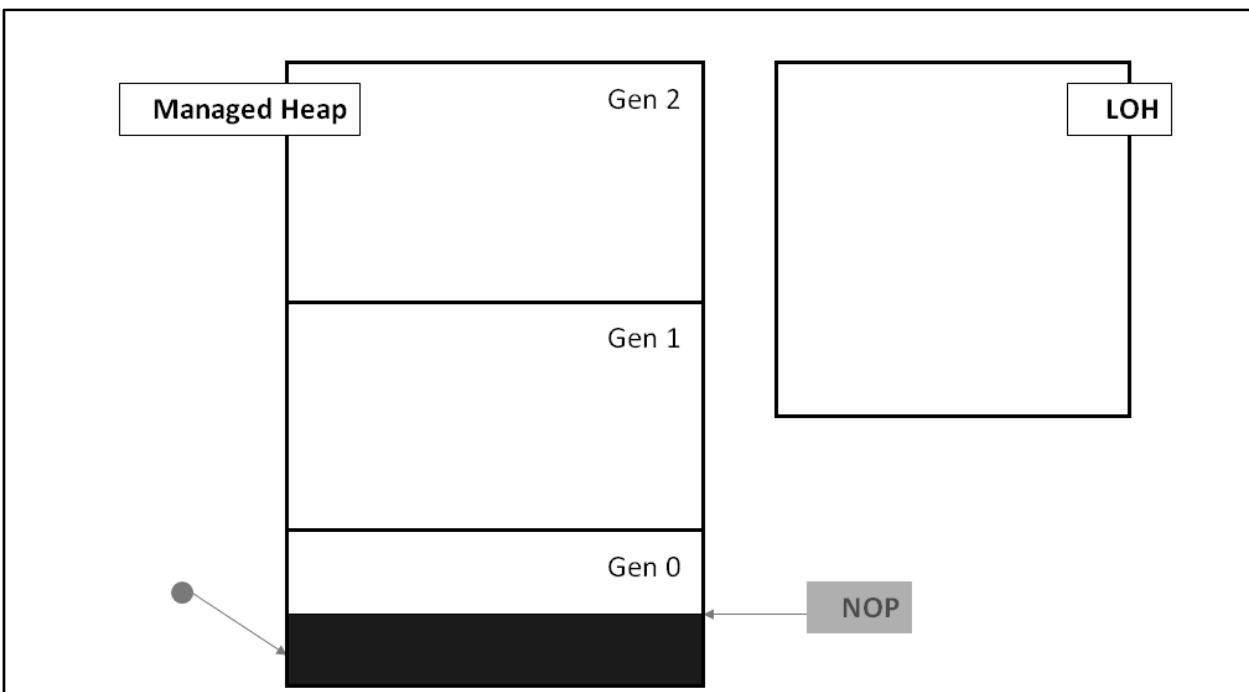
- Gen 0 and gen 1 collections are frequent but fast
- Gen 2 collections are slow but rare
- Large objects (>85KB) are managed in a separate Large Object Heap

The idea behind the generational model is that new objects tend to die quickly, so it is likely for a newly allocated object to be collected as part of a generation 0 or generation 1 collection and never reach generation 2. Because collecting the young generations is relatively cheap, most of the garbage collection cost is alleviated by separating objects by their age.

On the other hand, older objects which reach generation 2 are expected to survive for longer periods of time, so it does not harm the application's memory utilization to perform generation 2 collections only rarely.

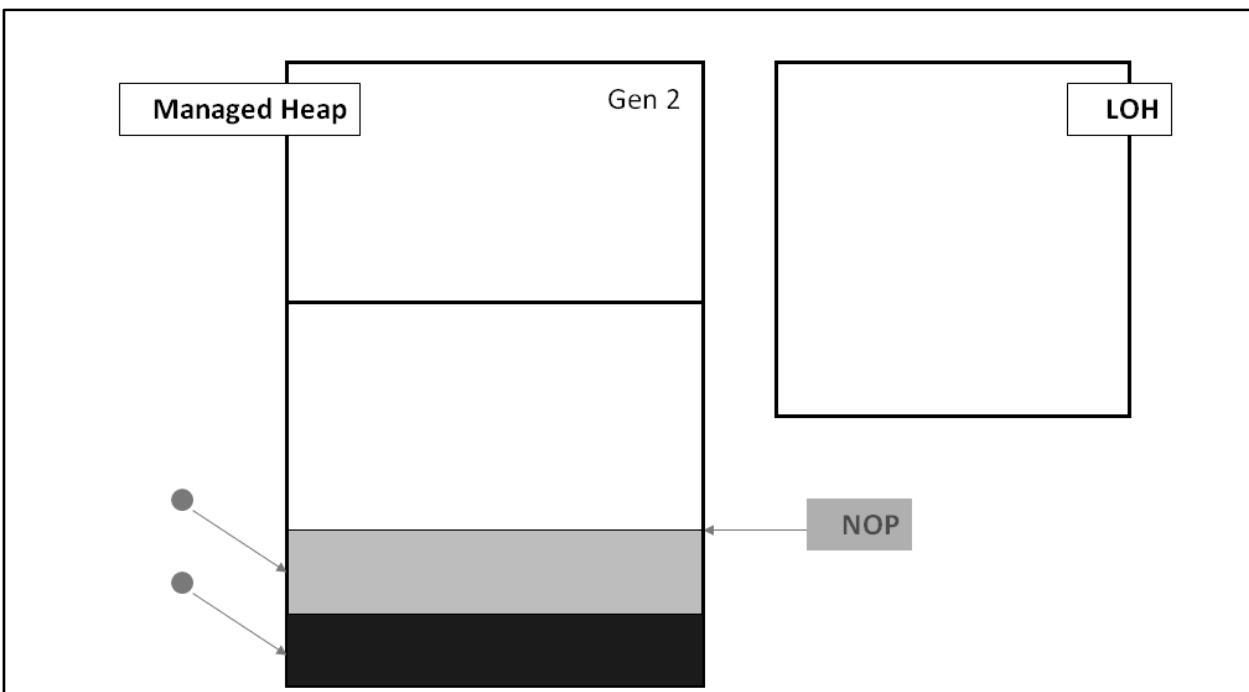
Finally, there is a separate region of memory for objects larger than 85KB (this is an arbitrary limit that is subject to change in future versions of the CLR). This differentiation ensures that large objects are not copied around as frequently. In the large object heap, the GC does not employ the compaction scheme described above within the sweep phase, but maintains a list of available memory chunks instead, similarly to the CRT allocator.

Generations Illustrated



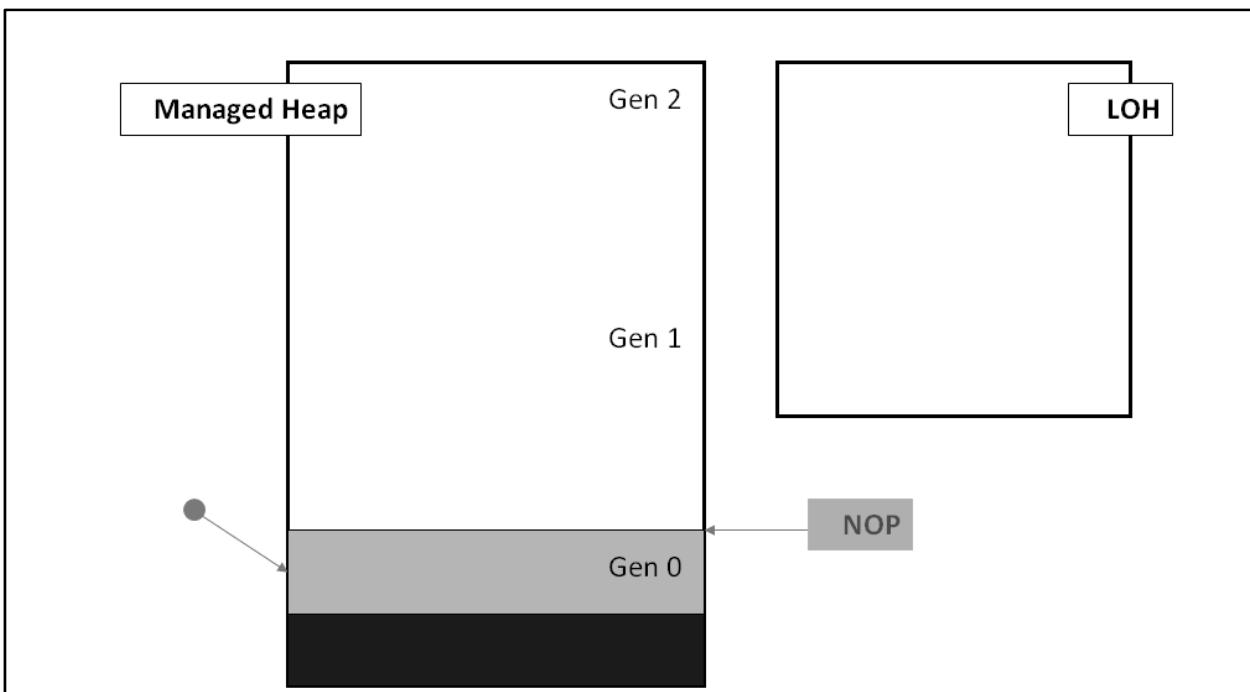
This slide demonstrates the situation in generation 0 before an allocation is made.

Allocations Are Made



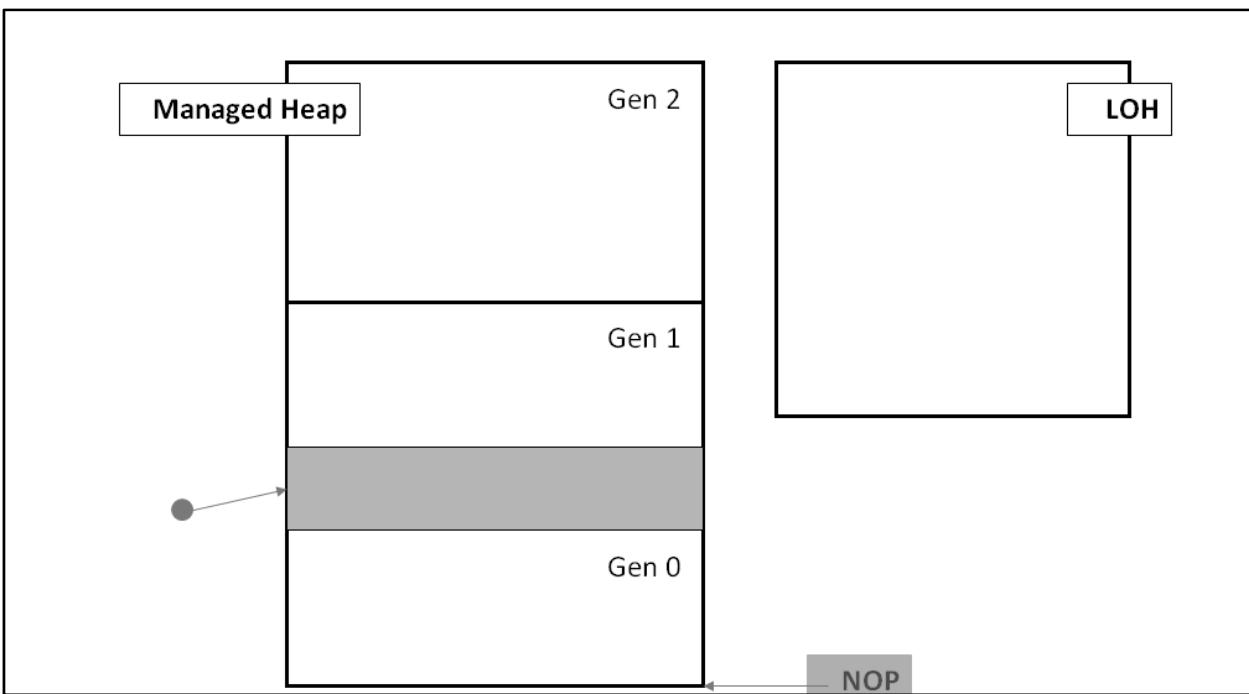
This slide demonstrates what happens when generation 0 fills. A GC is now required, but only in generation 0.

Generation 0 Fills



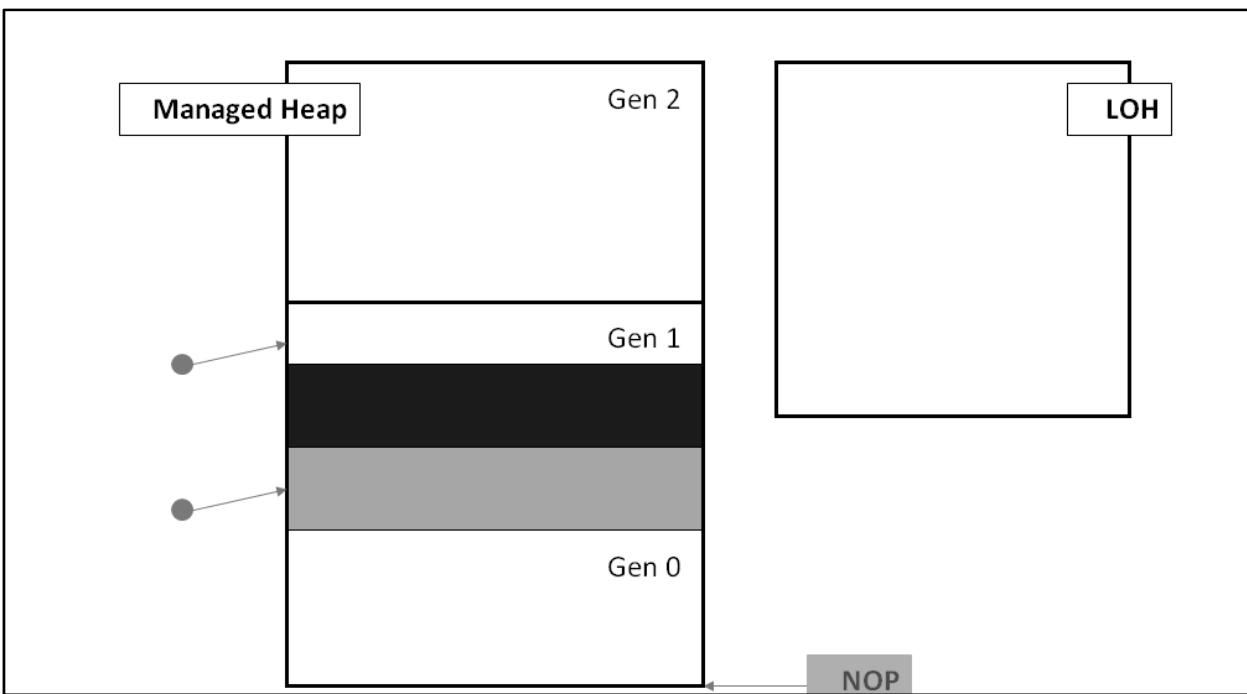
The lower object is unreachable, and will be reclaimed by the GC.

GC Occurs in Generation 0



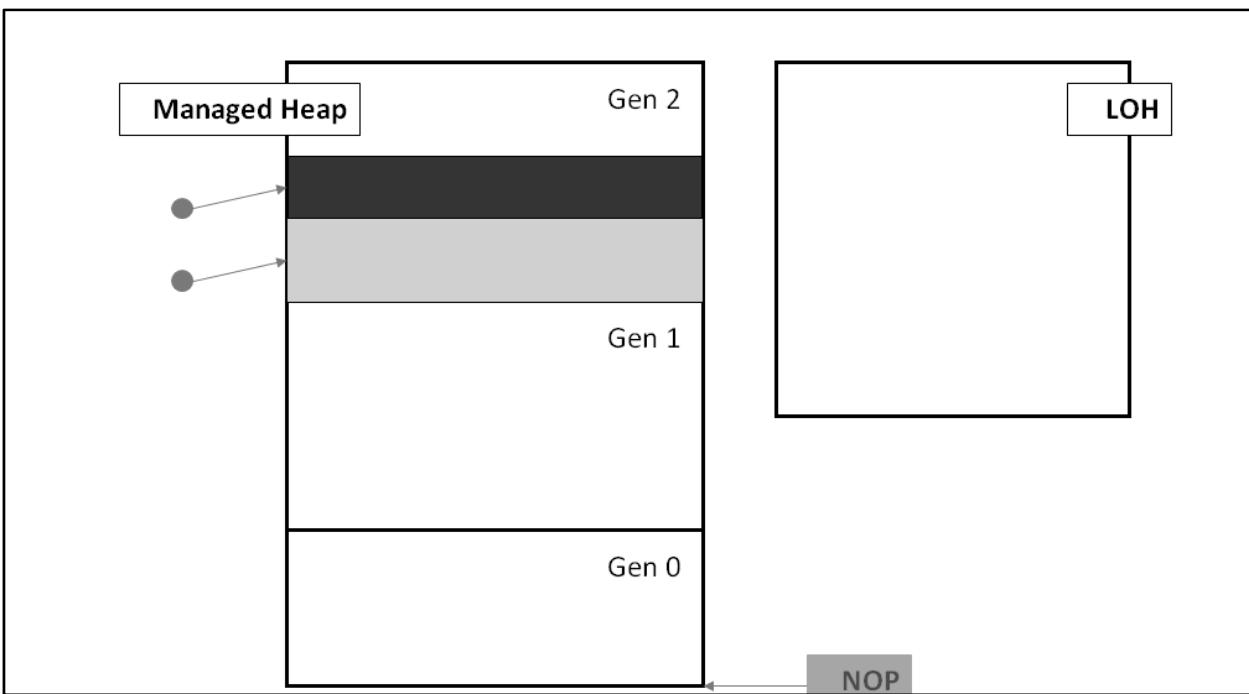
The surviving object is promoted to generation 1. Note that generation 0 is now empty, and the next object pointer points to the beginning of generation 0.

Generation 1 Fills



When generation 1 fills, a GC is in place for generation 1.

GC Occurs in Generation 1



Surviving objects from generation 1 are promoted to generation 2. After a collection in generation 1, it is also empty and ready to host survivors from generation 0.

Interacting with the GC

- The **System.GC** is the framework's static class which represents the GC

Informational Methods	Control Methods
GC.GetTotalMemory	GC.Collect
GC.GetGeneration GC.MaxGeneration	GC.AddMemoryPressure GC.RemoveMemoryPressure
GC.CollectionCount	GC.WaitForPendingFinalizers GC.SuppressFinalize GC.ReRegisterForFinalize

Interaction with the GC is performed primarily through the System.GC static class. Some methods provide information and some others provide control over the GC's actions. For the vast majority of applications, it is advised not to interfere with the GC's behavior and allow it to tune itself dynamically.

For advanced scenarios, the following methods might be of interest:

- GC.GetGeneration returns the generation to which an object belongs
- GC.GetTotalMemory retrieves the total memory used by managed objects in this application
- GC.Collect triggers a garbage collection in the specified generation
- GC.AddMemoryPressure and GC.RemoveMemoryPressure notify the garbage collector that unmanaged allocations are taking place within the host process, which might require the GC to tune itself accordingly.

Notification That a GC Occurs

- In .NET 3.5 SP1, a GC Notification API was added to the framework
- **GC.RegisterForFullGCNotification** and friends

Developers often want to know when a GC occurs, and even register to the notification programmatically. Theoretically, this enables low-latency scenarios in which an application can balance load or change its state if it becomes aware of a long-running GC that is about to occur. In practice, the use of these APIs is discouraged and is reserved for advanced scenarios. .NET 3.5 SP1 introduces a managed API that provides the ability to register for a GC notification and for a GC complete event. The following demo demonstrates this.



GC Notifications

See the **GC_Notifications** project in the **Module02_MemoryManagement_GC** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see the GC notifications API added in .NET 3.5 SP1 which allows an application to receive a notification when a GC approaches and when it completes. Please note that the use of the GC notifications API is possible only if not using concurrent GC. Because concurrent GC is the default, the demo uses an application configuration file (app.config) to toggle concurrent GC off using the `<gcConcurrent enabled="false" />` configuration element. NOTE: When using this demo, make sure to run in Debug mode so that the `byte[]` allocation is not optimized away. If it is, you can force a garbage collection by using the `GC.Collect` method (if running in Release mode).

Latency Mode and Collection Mode

- `GCSettings.LatencyMode`
 - `GCLatencyMode.Interactive`(WKS GC)
 - `GCLatencyMode.Batch` (SVR GC)
 - `GCLatencyMode.LowLatency`

- `GC.Collect(int, GCCollectionMode)`
 - `GCCollectionMode.Default`, `Forced`
 - `GCCollectionMode.Optimized`

Use with caution,
Preferably
within a CER

.NET 3.5 augments the managed GC API with several methods for controlling the GC behavior. The first is the `GCSettings.LatencyMode` enumeration, which takes the Interactive, Batch and LowLatency values. Interactive is equivalent to workstation GC, Batch is equivalent to server GC, and LowLatency is a new setting which enables a low-latency GC mode.

The low-latency GC mode means that the garbage collector will strive to minimize generation 2 collections while the mode is effective. Only if the system runs low on physical memory or if memory within the managed process is exhausted, a generation 2 collection will occur. This is an advanced setting that could be useful for low-latency scenarios, such as controlling a missile from managed code and requiring that a GC does not interfere with the missile-guiding code. However, low-latency GC does not mean a GC will not occur – it only means that the GC will make every effort to minimize the expensive garbage collections, in generation 2.

Not all state transitions are legal. For example, it's not allowed to switch the process' GC flavor from workstation to server at runtime. The only allowed transitions are between the Interactive and LowLatency modes, because the low-latency GC mode is supported only for workstation GC.

The second new feature is the overload of `GC.Collect` which takes a `GCCollectionMode` enum. The Default and Forced values are currently equivalent, meaning that the `GC.Collect` call will force a garbage collection and return. The Optimized value means that the GC is free to decide whether a garbage collection will be productive at the time of the call – and to disregard the call entirely if appropriate. This is the recommended value to pass if your application uses the `GC.Collect` API (which is discouraged in general).

Weak References

- A large object is rarely used
- Plan of action:
 - Allocate and keep alive
 - Allocate, use and destroy (repeat)

This scenario requires us to follow one of the two plans, but both have an unjustified worst-case cost.

If we opt to allocate the object and keep it alive, we pay the allocation cost once but the memory utilization cost is retained throughout the application's lifetime.

If we opt to allocate the object every time we need to use it and discard it afterwards, our memory utilization stays at a minimum but we pay the allocation cost (and the GC cost!) every time we need to use the object.

If it's unknown at compilation time how frequently the object will be used, it's hard to decide between the two alternatives (caching or not caching the object).

Weak References (contd.)

- Storing a *weak reference* to an object enables the GC to collect it
- A weak reference can be converted to a *strong one* if the object is alive
- Useful for any *service* that shouldn't keep the object alive

We can create a weak reference to an object using the `WeakReference` class and passing the object to its constructor. Subsequently, we can dispose of our strong reference to the object and keep only the weak one.

Later on, we can use the `WeakReference.IsAlive` and `WeakReference.Target` properties to determine whether the object is alive and to retrieve a strong reference to it if it is. (Note that this requires caution due to subtle race conditions – if `IsAlive` reports true, it doesn't mean that the `Target` won't be null when we retrieve it. So first we have to retrieve the target, then check if it's not null, and only then use the reference.)

Note that this is not necessarily applicable for our scenario. If we keep a weak reference to the expensive object, then it will be reclaimed the first time a GC occurs – and if a GC occurs frequently, there's hardly any difference between this and the second case (allocating the object anew every time). However, we could dynamically decide whether we want to store a strong or a weak reference based on some other data in the program, such as available memory or the number of garbage collections.

Weak references are generally useful for any service that is provided to an object and should not keep the object alive, such as timers, event mechanisms etc. For example, it's highly likely that we don't want an event handler to stay alive if there is no other reference to it from the application other than the event source. This kind of scenario can be enabled through custom interfaces and weak references. In the lab, you will experiment with a weak-reference-based timer, which does not keep the object alive if it is otherwise unreachable by the application. This technique can minimize memory leaks and remove the burden from the programmer's shoulder (remembering which mechanisms might keep an unwanted to our objects).



Weak References: Cache

See the **WeakReferences_Cache** project in the **Module02_MemoryManagement_GC** solution under the SampleCode folder for more information about this demo in the code comments. A cache that caches strong references to some percent of the objects and weak references to some other percent of the objects on a first-come-first-serve basis. I.e., older objects are retained through strong references and when the strong limit is exhausted, new objects are added as weak references.

Finalization

- Unmanaged resources require finalization code
- A finalizer (`~ClassName`) is mapped to a method overriding `Object.Finalize`
- Finalization code is executed *at some point* after the object becomes unreachable

Not all resources referred to in a program can be completely managed by the CLR. For example, while the GC has no problem determining when memory should be reclaimed, it does not have intimate knowledge about resources such as files (represented by operating system handles), database connections, sockets and many others. Knowing when to free these resources, and more importantly *how* to free them, remains the application's responsibility and not the runtime's.

Nonetheless, the CLR in general and C# in particular provide a way to specify that finalization code is associated with a particular resource. This is done by writing a special method (which is an override of `Object.Finalize`) called a finalizer, which has the same name as the class but preceded by a tilde sign (~). The code within that method will be executed *at some point in time* after the object has become unreachable. Note that the CLR does not provide a deterministic finalization guarantee.

Finalization Internals

- Finalizable objects start in the **finalization queue**
- When GC finds the object unreachable it is moved to the **freachable queue**
- The **finalizer thread** executes its finalization method
- At the next GC, the object is reclaimed

When an object is created, the CLR checks to see if it has a finalizer. If it does, a reference to the object is added to a special runtime-managed queue called the *finalization queue*, which serves as a root as far as the garbage collector is concerned.

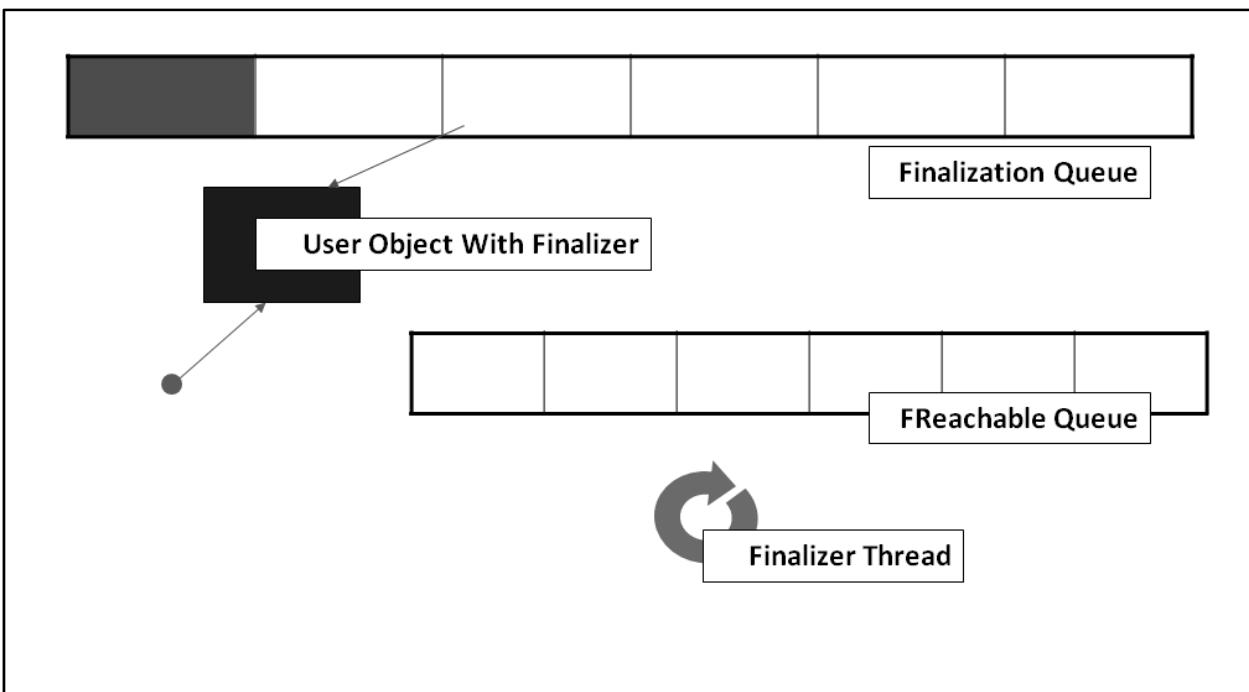
When the object becomes unreferenced from the application, the GC makes note of the fact the last reference to the object is from the *finalization queue* and moves it to another runtime-managed queue called the *freachable queue*. (Note that this occurs only when a GC occurs – there is no other process that can deem an object as waiting for finalization.)

Next, the *finalizer thread* wakes up (at some point) and executes the finalization methods of all objects that are in the freachable queue. These objects are removed from the freachable queue.

When the next garbage collection occurs, the object is no longer referenced from the application or from the finalization queues, and therefore its memory is reclaimed.

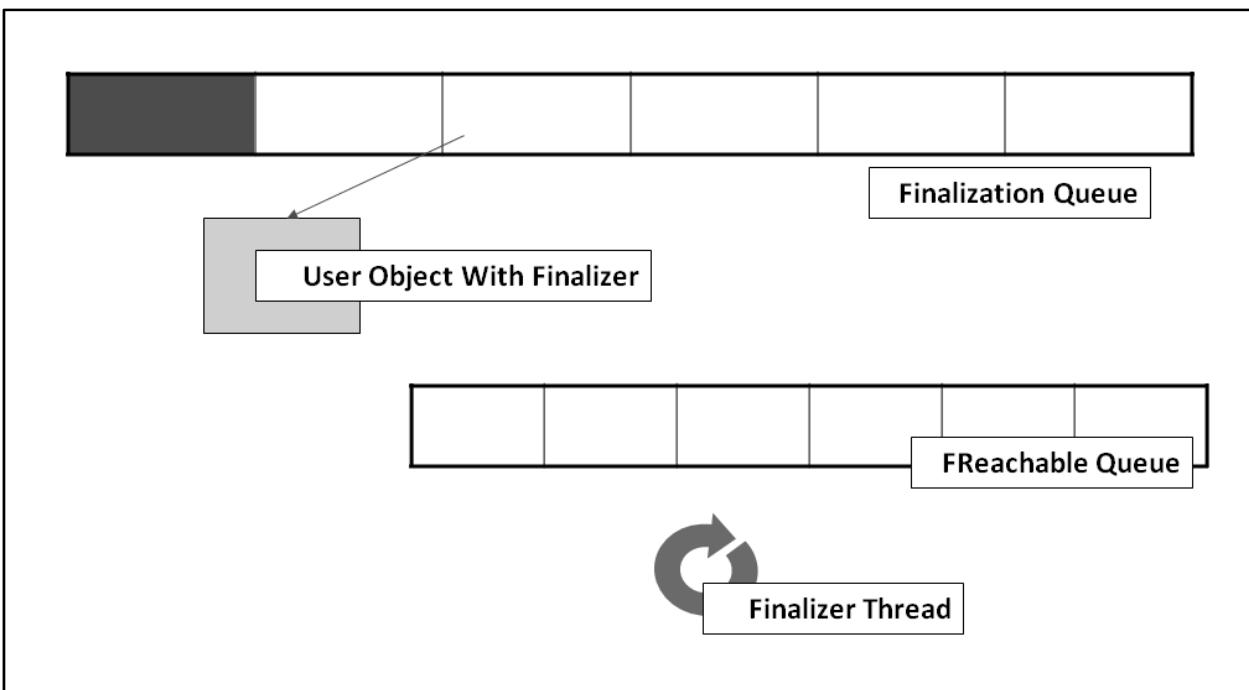
Nonetheless, it survived at least two garbage collections, and therefore was promoted at least to generation 1. The degree of pressure this puts on the runtime and on the application's memory utilization is not negligible, which is a significant reason to avoid finalization altogether.

Finalization Illustrated



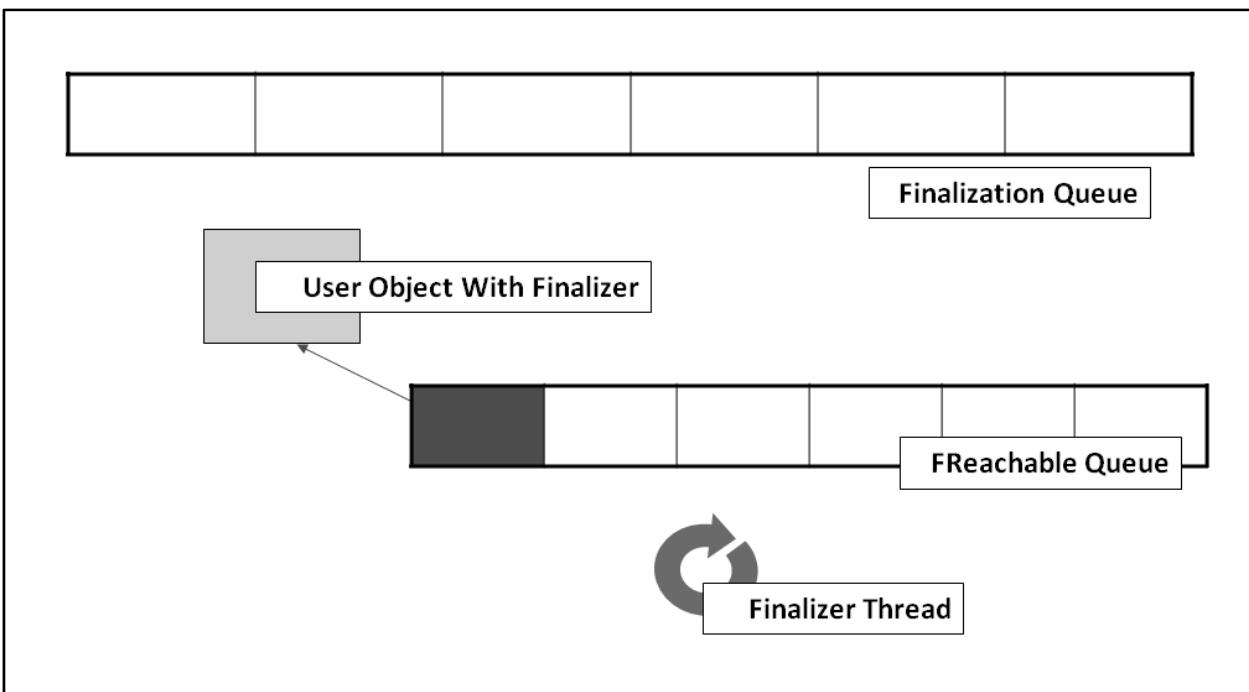
An object with a finalizer is created, and a reference to it is automatically added to the runtime-managed finalization queue.

Object is No Longer in Use

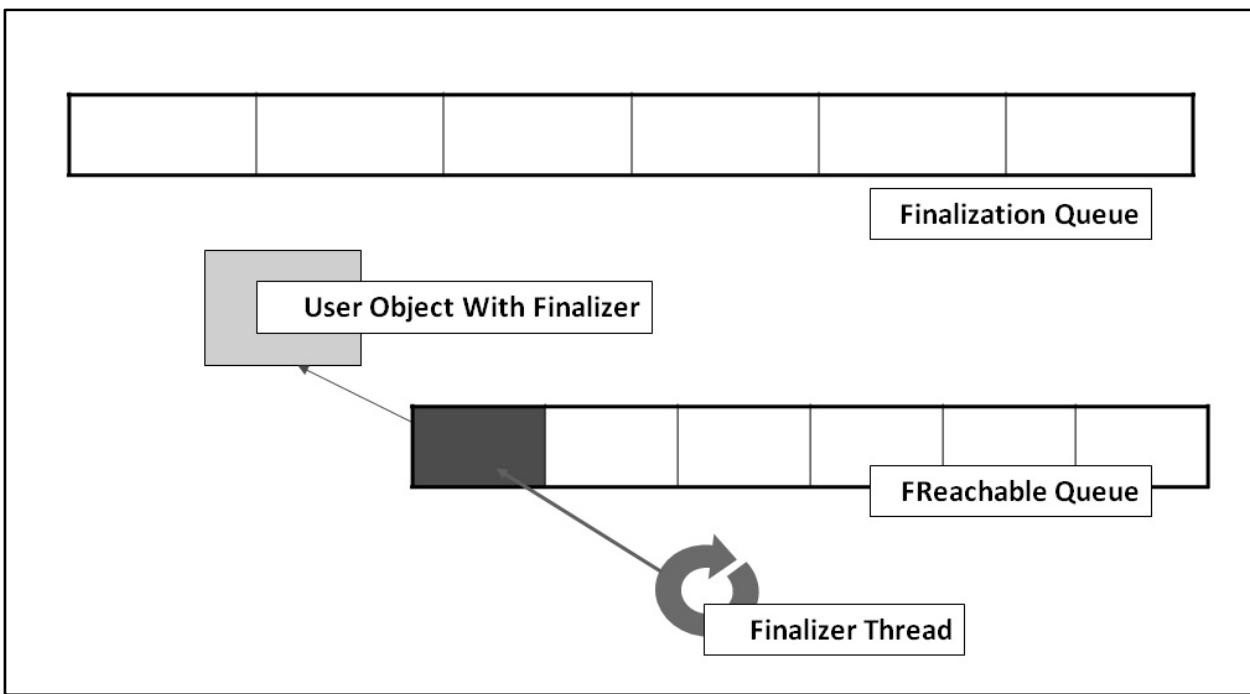


The object is no longer reachable by the application. Now a garbage collection occurs...

GC Occurs



The GC moves the reference to the object from the finalization queue to the freachable queue.

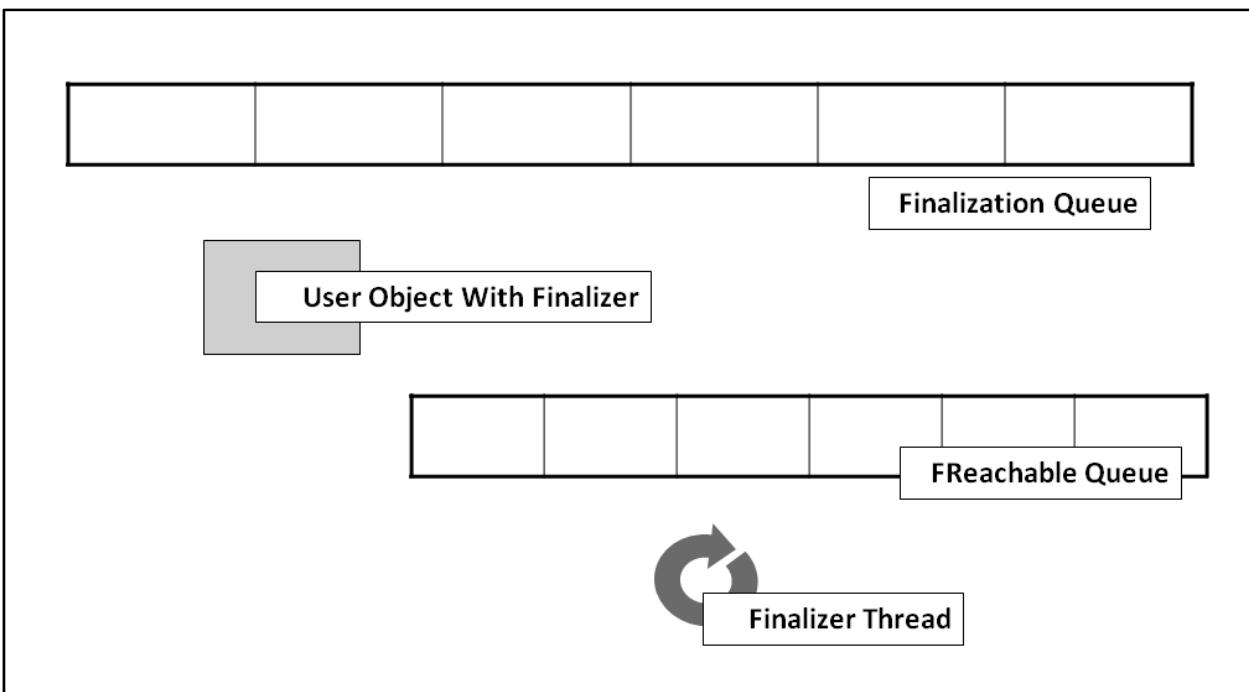
Finalizer Thread Wakes Up

The finalizer thread wakes up and executes the object's finalizer.

(Note: the finalizer thread does not wake up for every object. Its heuristics are subject to change with every version of the CLR.)

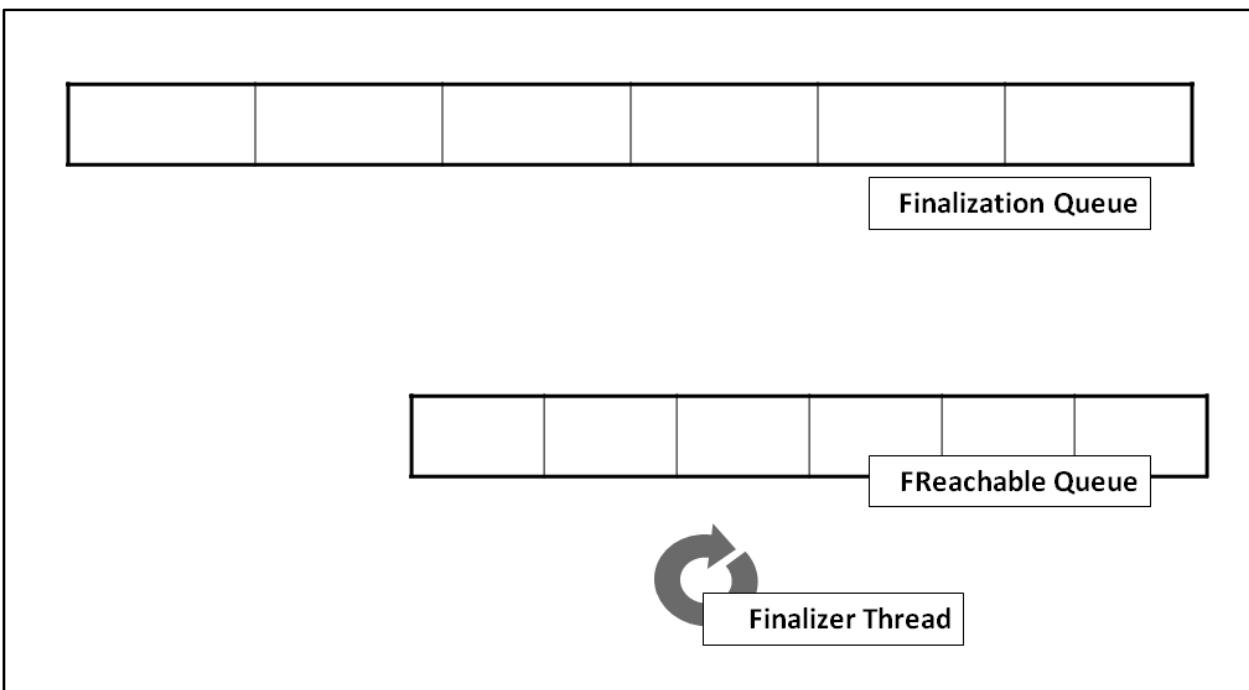
2-42 | Module 02 - Memory Management (GC)

Finalizer Is Done



The finalizer is done and removes the reference to the object from the freachable queue. The object is now unreachable.

Another GC Occurs



The next garbage collection cycle cleans up the object's memory as it is no longer reachable.

Finalization Pitfalls: Memory Leaks, Race Conditions, Deadlocks



See the **FinalizationPitfalls** project in the **Module02_MemoryManagement_GC** solution under the SampleCode folder for more information about this demo in the code comments.

The pitfalls demonstrated by the demo are:

- Memory leak caused by a faster allocation rate than finalization rate
- Deadlock caused by application code which acquires a lock and waits for the finalizer which needs the same lock
- Race condition caused by application code initiating a lengthy operation while at the meantime underlying resources are destroyed by the finalizer

Avoid Finalization If Possible

- Finalization has terrible performance
- Finalization can cause correctness problems



How can we obtain deterministic finalization?

.NET default finalization has a significant performance cost and can easily lead to many bugs due to its non-deterministic and asynchronous nature. What is the alternative, however? How can we obtain deterministic finalization?

We already know the answer – we must do it explicitly, on the application level, without any runtime support.

The Dispose Pattern

- Implement **IDisposable**
- A single method called **Dispose**
- Perform finalization work in that method
- To prevent double deletion (and effect on the finalization performance), call **GC.SuppressFinalize**

The Dispose pattern is a standardized way of implementing deterministic finalization in an application (although any other way might work as well). The pattern consists of an interface called **IDisposable** which has a single method called **Dispose**. This method is called by the class' client by convention when the client has finished using the object. In its implementation of **Dispose**, the class performs deterministic finalization work, and when it's done, it would usually mark itself disposed using a boolean flag and throw an **ObjectDisposedException** exception if the client attempts to use the same instance again.

Oftentimes, a class with a **Dispose** method will still have a finalizer to ensure that even if the client forgot to call **Dispose** (e.g. under exceptional conditions), the resources will still be cleaned up. To ensure that the finalizer isn't called after the **Dispose** method already performed deterministic finalization and to avoid the performance penalty associated with finalization book-keeping, the **Dispose** method should call **GC.SuppressFinalize** passing *this* as the parameter to let the GC know that this instance does not require finalization.

Note that clients should use *try...finally* or *using* blocks to ensure that every resource that implements **IDisposable** is promptly disposed of. Relying on deterministic finalization on the client side is error-prone.

Lab: Weak Timer



Weak Timer Lab:

Use the **WeakTimer_Starter** solution under the **Exercises\Module02_MemoryManagement_GC** folder as a starter solution for implementing a weak-reference-based timer.

Instead of relying on a delegate, this timer implementation will use an interface that the subscribing objects implement. The timer will keep a weak reference to this interface and detect that the weak reference has been collected to automatically cancel the timer if necessary.

This kind of timer prevents memory leaks and unnecessary timer invocations when the original registered object is no longer reachable from the main application code.

You can find the solution for this lab in the **WeakTimer_Solution** solution under the

ExerciseSolutions\Module02_MemoryManagement_GC folder.

Summary

- Overview of memory management
- Garbage collection first steps
- GC flavors
- Generations
- Interacting with the GC
- Weak references
- Finalization and Dispose
- Lab

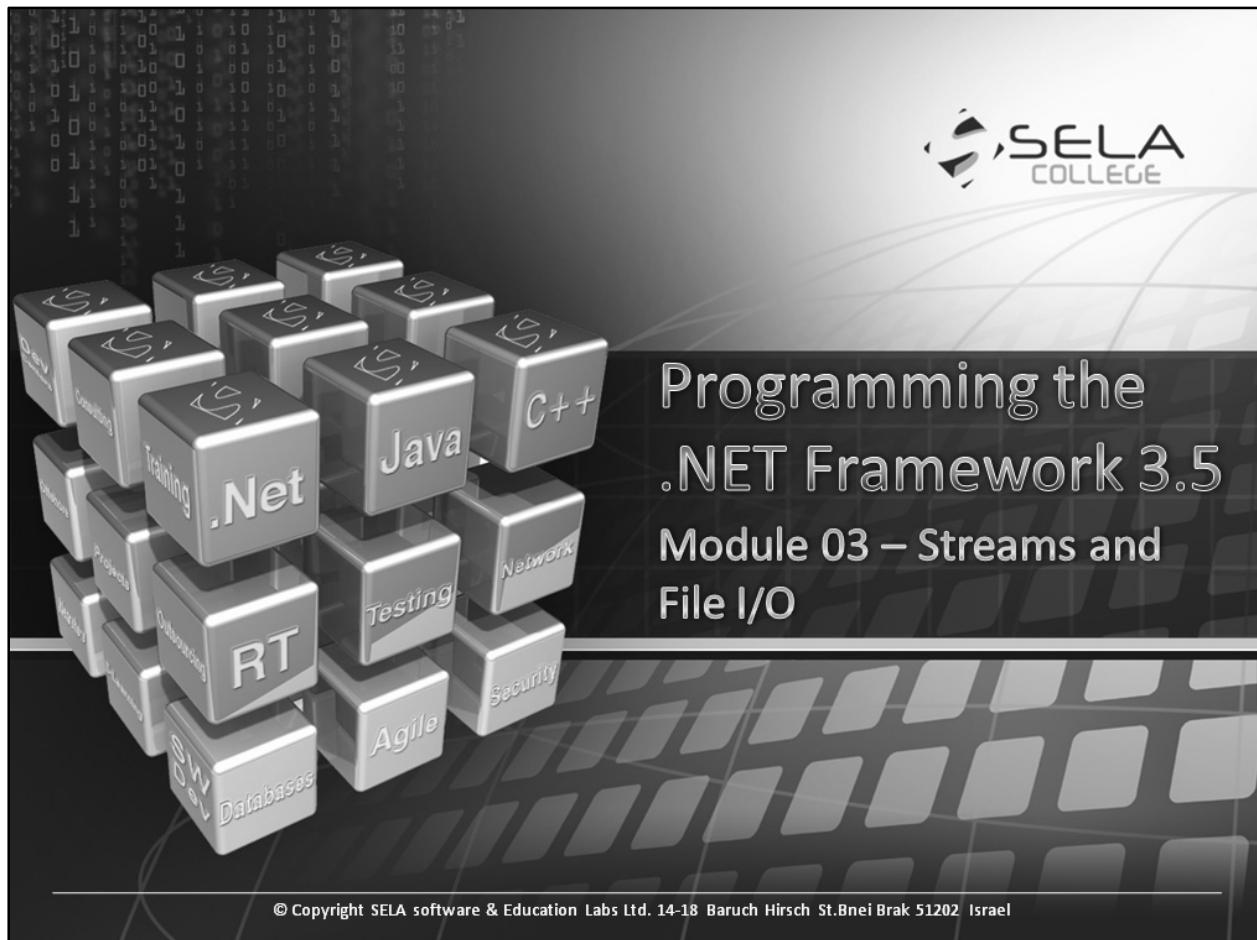




Module 03 - Streams and File IO

Contents:

In This Chapter.....	3
Input and Output Abstraction	4
Streams Direct Hierarchy	5
Decorators and Composites.....	6
File Streams	8
Working with File Streams	9
Stream Readers and Writers.....	10
Working with Readers / Writers	11
Binary Readers and Writers	12
Writing Binary Data	13
Reading Binary Data	14
File and Directory Classes	15
Lab: Word Count	17
Summary	18



In This Chapter

- ⌚ Streams as data abstraction
- ⌚ File streams
- ⌚ Stream readers/writers
- ⌚ The File and Directory classes
- ⌚ Lab



MCT USE ONLY. STUDENT USE PROHIBITED

Input and Output Abstraction



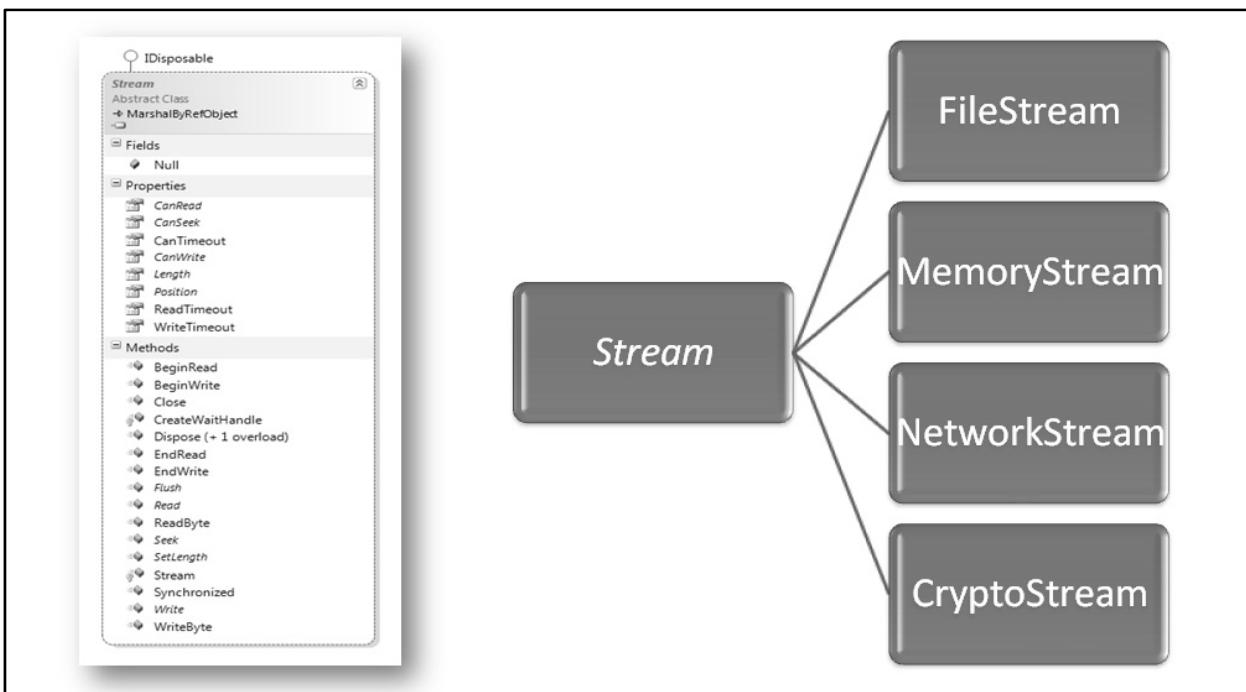
What's common to all input and output sources?

- Stream
 - Length, position, content-type
 - Read, write, seek
 - Open, close

This is actually not a very easy question to answer. Input and output tend to differ greatly from one another, and the data that is read or written can be presented in a huge variety of different forms. What is usually common to all I/O abstractions is the notion of a stream of data, which sometimes is irreversible (e.g. you can't retract bytes sent over the network) and at other times can be accessed randomly (e.g. local files).

A stream of data can have lots of useful properties, such as length, position, content type; and can have lots of useful methods such as read, write, seek and others. The .NET abstraction of streams is no different in that it provides an abstract base class called `Stream` which is the root of the inheritance hierarchy for all kinds of input and output sources.

Streams Direct Hierarchy



The direct hierarchy of streams does not do justice to the richness of the .NET stream-related libraries. The reason is that quite a significant amount of functionality is exposed via decorators and adaptor patterns which use an underlying stream in their implementation. A stream such as the ones in this diagram provides only the most rudimentary facilities with regard to its underlying data source, such as writing or reading an array of bytes. No advanced data manipulation or operations are supported at this level.

Decorators and Composites

- Streams can be built on top of streams
- Aggregation, inheritance

A common decorator or composite pattern can be accomplished by aggregating streams within other streams.

For example, the `CompressedStream` and the `CryptoStream` do not offer I/O functionality on their own – they wrap another underlying stream and add a level of decoration to it. Inheriting a stream is also a possibility because the key methods are marked `virtual` as needed.

We can design our own decorated and composite streams, which is the subject of the following demo.



See the **StreamsAndDecorators** project in the **Module03_Streams_FileIO** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you'll experience the power of decorators and inheritance with regard to the stream-oriented data abstraction. Our first abstraction is the CompositeOutputStream class, which accepts multiple streams in its constructor and performs all output (write) operations on all streams at once. The CompositeOutputStream instances act as a stream as well (thanks to inheritance from the abstract Stream class), thus playing along with the abstraction. The demo of this feature shows how a write can be automatically directed to the standard error and standard output streams at once.

Our second abstraction is the XorEncryptionStream class, which performs a trivial XOR encryption and decryption of data that passes through the stream. For the ease of developing other decorators on a stream which delegate most functionality to the underlying stream, the DecoratingStream abstract class implements this delegation. The XorEncryptionStream class overrides only the Read and Write methods to encrypt and decrypt the data with a simple byte key taken as a parameter. (Note that by no means is XOR encryption sufficient for a secure application; this is used only in the context of this demo for expository purposes.)

File Streams

- File streams enable file I/O
- File name
- File access: Read, write, both
- File open: Create, truncate
- File share: None, read, write, delete

File streams are the implementation of the stream abstraction for file input and output operations, and reside in the System.IO namespace. A file stream has several special characteristics that a general stream doesn't have, these include:

- A file name, which can be any UNC path as well as a local file name.
- File access mode, which can be read, write or both read and write (this primarily affects security, as you won't be able to open for writing a file that you only have read permissions for).
- File open mode, which can take values like create, create new, truncate, open existing etc. – this mainly affects the API behavior in case the file exists, in case it doesn't exist, what to do when opening an existing file and so forth.
- File share mode, which can take values like read, write, delete, none – indicating what other processes can do with the file while this process is using it. For example, when opening a file for read-only access, there is usually no reason to prevent other processes from accessing it for read as well.

Working with File Streams

```
1 FileStream myFile = new FileStream("myfile.txt",
2     FileMode.Create, FileAccess.ReadWrite);
3 string greeting = "Good afternoon";
4 byte[] output = Encoding.Unicode.GetBytes(greeting);
5 myFile.Write(output, 0, output.Length);
6 myFile.Close();
7
8 myFile = new FileStream("myfile.txt",
9     FileMode.Open, FileAccess.Read, FileShare.Read);
10 byte[] input = new byte[myFile.Length];
11 myFile.Read(input, 0, input.Length);
12 myFile.Close();
13 Console.WriteLine("Read from file: " +
14     Encoding.Unicode.GetString(input));
```

A large, stylized, three-dimensional text logo where each letter 'C', 'O', 'D', and 'E' is composed of multiple overlapping rectangular planes, creating a sense of depth and perspective.

This example demonstrates quite a large amount of functionality around file streams, including reading from files, writing to files, obtaining the stream's length and converting arrays of bytes into legible strings using the Encoding class.

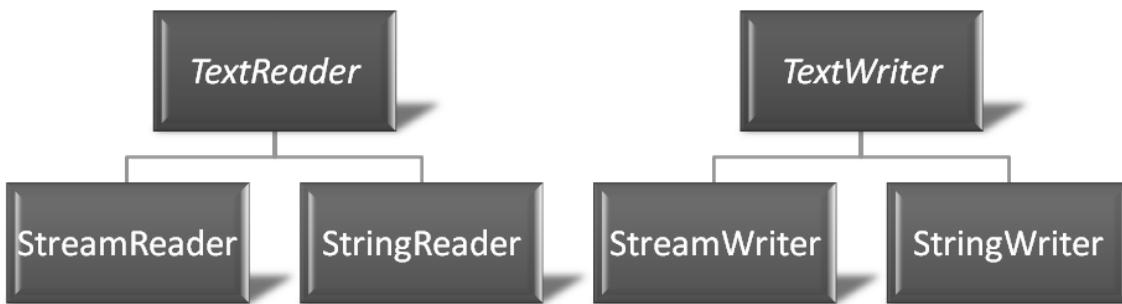
Lines 1-6 contain code which opens a file for read and write access, and proceeds to write to it the greeting string encoded as Unicode. Subsequently, the code closes the stream (line 6) to ensure that the stream is flushed and its associated resources are released. (Note that even though the FileStream class has a finalizer, as we have learned it is still not advisable to rely on non-deterministic finalization if we know for sure that we're done using an object.)

Lines 8-14 contain code which opens the file for read access only, and therefore doesn't mind sharing the file with other readers. The code then determines the stream's length, allocates an array that would suffice to contain the stream's contents and reads the data from the stream. Subsequently, it closes the stream and displays the string read using the Unicode encoding.

3-10 | Module 03 - Streams and File IO

Stream Readers and Writers

- Working with streams is cumbersome
- Readers and writers simplify work
- Writers buffer data



Stream readers and writers provide functionality for reading strings from a stream and writing strings to a stream, and generally offer a less frightening API. Note that the inheritance hierarchy includes a sibling *StringReader* and *StringWriter* which offer string-oriented functionality similar to what *StreamReader* and *StreamWriter* offer on top of a stream. *StreamReader* and *StreamWriter* also provide a public property called *BaseStream* which allows direct manipulation of the underlying stream (as the readers and writers do not have independent functionality without an underlying stream). This abstraction allows the same *StreamReader* and *StreamWriter* classes to be used with every type of stream – *MemoryStream*, *FileStream* and any other exotic derivation from the abstract *Stream* class. These are the text-oriented readers and writers; there are also binary-oriented readers and writers which we will look into shortly.

The *StreamWriter* class performs buffering of data so that fewer I/O requests are issued. This usually improves the performance of I/O operations, but in rare cases when it doesn't there are ways to cause the buffer to flush immediately (e.g. using the *Flush* method). Even if you do not care about the performance of I/O operations in your program, you should be aware to the buffering – not every write to a *StreamWriter* object is immediately propagated to the underlying stream.

Working with Readers / Writers

```
1 FileStream myFile = new FileStream("myfile.txt",
2 FileMode.Create);
3 StreamWriter writer = new StreamWriter(myFile);
4 writer.WriteLine("From a stream writer!");
5 writer.Close(); //This is critical!
6
7 //Short-hand for creating a file
8 StreamReader reader = new StreamReader("myfile.txt");
9 Console.WriteLine(reader.ReadLine());
10 reader.Close();
```

TIP

StreamWriter does
not have a finalizer
Close is critical

Binary Readers and Writers

- Oriented towards binary data
- Methods for various data types (not just strings)

The `BinaryReader` and `BinaryWriter` classes are very similar in nature to the `StreamReader` and `StreamWriter` classes (e.g. all of these classes incorporate a wrapped stream through a `BaseStream` property), but they are oriented towards reading and writing binary data. The reader/writer classes have a variety of methods for reading and writing all kinds of managed data types, including all the primitive types as well as arrays.

When the need for custom parsing of binary data or custom serialization to binary data arises, the `BinaryReader` and `BinaryWriter` classes offer a very good selection of methods for streamlining the process. In the following demo slide, we will see an example of matrix data serialization to a binary file.

Writing Binary Data

- Note that the matrix bounds are serialized

```
1 BinaryWriter writer = new BinaryWriter(  
2     new FileStream("matrix.dat", FileMode.Create));  
3 writer.Write(matrix.GetUpperBound(0));  
4 writer.Write(matrix.GetUpperBound(1));  
5 for (int i = 0; i < 10; ++i)  
6     for (int j = 0; j < 10; ++j)  
7         writer.Write(matrix[i, j]);  
8 writer.Close();
```

CODE

In this example, a `BinaryWriter` is created wrapping a `FileStream`. Next, the matrix outer and inner dimensions are serialized using calls to the `Write` overload which accepts an `Int32`. Next, the matrix itself is serialized, and finally the writer is closed, which also closes the underlying stream. We're using two different overloads of the `Write` method here (for `Int32` and for a `Double`) and there are overloads suitable for all primitive types, for arrays and for strings. Note that the matrix bounds are serialized to the stream here because when we open the file and start deserializing the data, we will need the matrix bounds to know what array to allocate. Writing the data bounds to the binary file before the actual data is a fairly common practice with file-based databases, network protocols and a variety of other applications.

3-14 | Module 03 - Streams and File IO

Reading Binary Data

- ➊ The matrix bounds are read first

```
1 BinaryReader reader = new BinaryReader(  
2     new FileStream("matrix.dat", FileMode.Open));  
3 int dim0 = reader.ReadInt32();  
4 int dim1 = reader.ReadInt32();  
5 double[,] newMatrix = new double[dim0, dim1];  
6 for (int i = 0; i < dim0; ++i)  
7     for (int j = 0; j < dim1; ++j)  
8         newMatrix[i, j] = reader.ReadDouble();  
9 reader.Close();
```

CODE

In this example, a `BinaryReader` is created wrapping a `FileStream` again. The reader assumes prior knowledge about the structure of the binary file, and therefore reads the two matrix dimensions first, subsequently creating and filling the matrix. Note how each read is accompanied by a different method call – `ReadInt32` for `Int32`, `ReadDouble` for `Double` and so on. Convenience methods exist for most primitive types, arrays and strings. Eventually, the reader is closed, releasing the file and the reader.

File and Directory Classes

```
1 void FillTreeViewHelper(string path, TreeNode node)
2 {
3     foreach (string dir in Directory.GetDirectories(path))
4     {
5         FillTreeViewHelper(dir, node.Nodes.Add(dir));
6     }
7     foreach (string file in Directory.GetFiles(path))
8     {
9         currentNode.Nodes.Add(file);
10    }
11 }
```

- Helper classes for common operations.
- File and directory information.

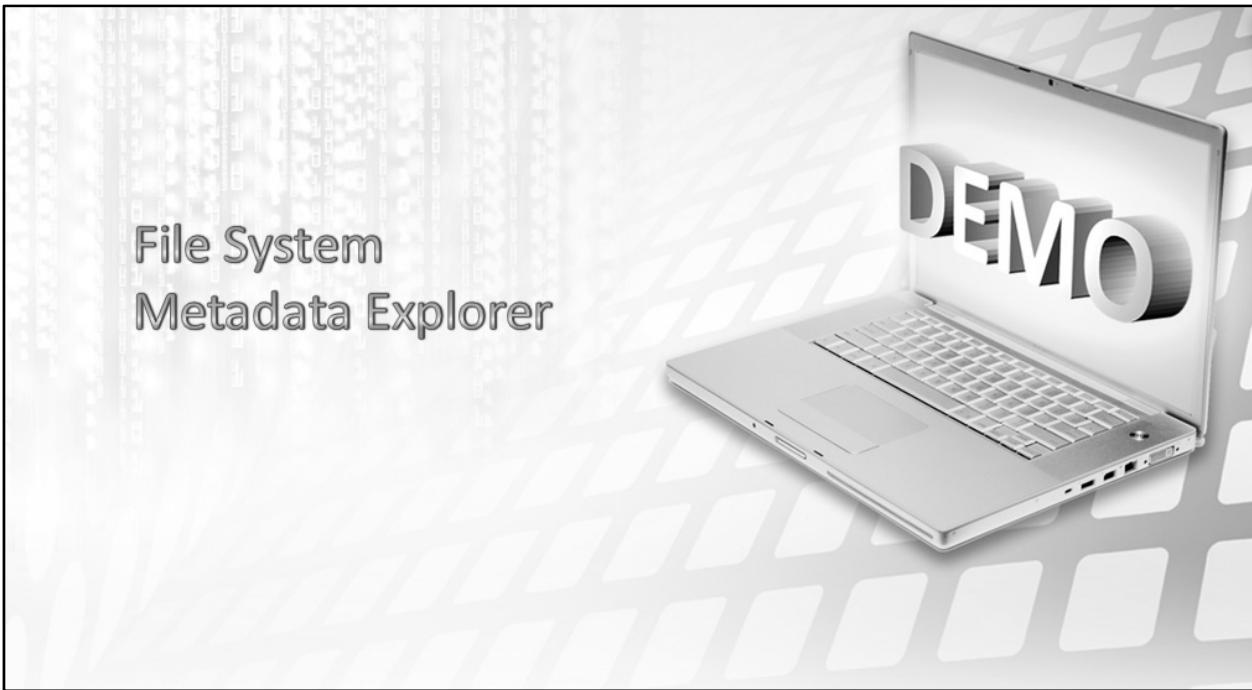
The File and Directory classes (under System.IO) facilitate working with file streams and file-based stream readers and writers. Methods such as File.OpenText, File.WriteAllLines and others are extremely useful for performing common operations on files, especially common text-based operations. Methods such as Directory.GetFiles offer quick insight into the directory structure of the file system.

However, there are also the FileInfo and DirectoryInfo classes which provide an insight into the file and directory metadata instead of their contents. For example, the FileInfo class does not provide methods for reading or writing the file contents, but provides access to the file's attributes, creation time, security permissions and other information.

The code example on this slide demonstrates a recursive method which uses the Directory.GetDirectories and Directory.GetFiles methods to fill a visual tree view with directories and files corresponding to the file system hierarchy.

The following demo will demonstrate some of the File, FileInfo, Directory and DirectoryInfo functionality spectrum.

3-16 | Module 03 - Streams and File IO



See the **FileSystemMetadataExplorer** project in the **Module03_Streams_FileIO** solution under the SampleCode folder for more information about this demo in the code comments.

This demo is a UI application which displays the hierarchical file system structure using a tree view control. The construction of the tree shows how a simple recursive algorithm working over all files and directories under a given path (using the `Directory` class) can be implemented.

The other part of the UI shows specific file details. The preview section demonstrates how the shortcut-enabling `File` class APIs can retrieve all text in a given file. The property grid shows metadata directly obtained from `DirectoryInfo` and `FileInfo` objects for directories and files respectively.

Lab: Word Count



Word Count Lab:

Use the **WordCount_Starter** solution under the **Exercises\Module03_Streams_FileIO** folder as a starter solution for implementing an application that counts the number of words in a set of files.

The application outputs the word counting results to a text writer (which defaults to the standard output stream) and to a binary file containing only the word counts in each file. It parses command line parameters by using the supplied starter code.

You can find the solution for this lab in the **WordCount_Solution** solution under the **ExerciseSolutions\Module03_Streams_FileIO** folder.

3-18 | Module 03 - Streams and File IO

Summary

- Streams as a data abstraction
- File streams
- Stream readers / writers
- The File and Directory classes
- Lab



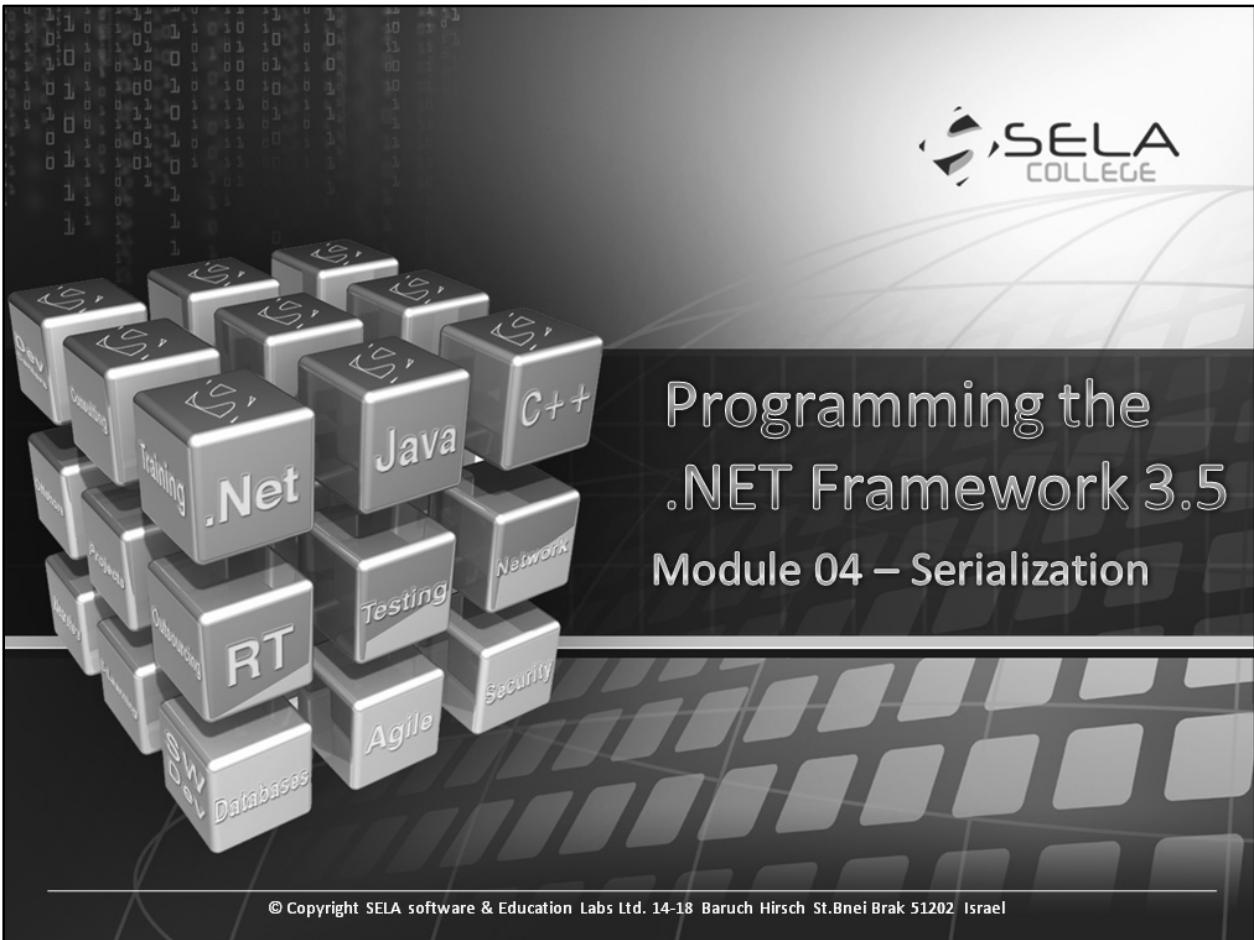


Module 04 - Serialization

Contents:

In This Chapter.....	3
What's Serialization?.....	4
Why Serialization?	5
.NET Serialization	6
Serializable User Class.....	7
BinaryFormatter	8
Controlling Serialization	10
Serialization Callbacks	11
Custom Serialization	13
Custom Serialization Example	14
Serialization Alternatives.....	15
Lab: Serialization Framework	17
Summary	18

4-2 Module 04 - Serialization



Mar 12 2010 7:47PM Ivan Kirkorau ivan_kirkorau@epam.com

In This Chapter

- ⌚ Motivation for serialization
- ⌚ Marking a type for serialization
- ⌚ BinaryFormatter
- ⌚ Controlling serialization
- ⌚ Custom serialization
- ⌚ Overview of XML serialization
- ⌚ Overview ofDataContract serialization
- ⌚ Lab



4-4 | Module 04 - Serialization

What's Serialization?

- *Serialization is the process of saving an object onto a storage medium (such as a file, or a memory buffer)...*

<http://en.wikipedia.org/wiki/Serialization>

This is the Wikipedia definition of serialization. In simpler words, serializing an object means taking the object and representing it as a series of bytes. These bytes can be stored persistently, sent over the network, dumped away – the point is that there is a way to construct a series of bytes from an object. This series of bytes would be worthless if there were no way to restore the object from them – and this is what deserialization is about. Deserializing an object means taking the series of bytes that represents the object and creating from it a populated object instance.

Serialization and deserialization can be performed in a variety of ways. For example, we could envision a textual representation of the object's data, an XML representation, a binary representation, a compressed representation and so on. The .NET framework features facilities for all kinds of serialization mechanisms, and we will examine them in detail in this module.

Why Serialization?

- Persistence
- Communication
- Logging

But first, why do we need serialization at all? What's the value?

Serialization is suitable for at least the following scenarios:

- Persistence – when an application uses information that has a lifetime longer than the application's OS process, it will need to use serialization so that these objects can be persisted to persistent storage (database, disk etc.) and loaded from that storage later.
- Communication – when two applications which do not share the same OS process (or even the same .NET AppDomain) need to pass objects between them, the object reference can't be passed directly. The only way to safely transmit an object's data is by serializing it to a stream of bytes and deserializing it at the other end. (This is what .NET remoting is all about, and we will discuss it in brief in the AppDomains chapter.)
- Logging – serializing data about objects in the application is extremely useful for debugging and diagnostics scenarios. There is no better way to obtain information about an object than by looking at the object's contents.

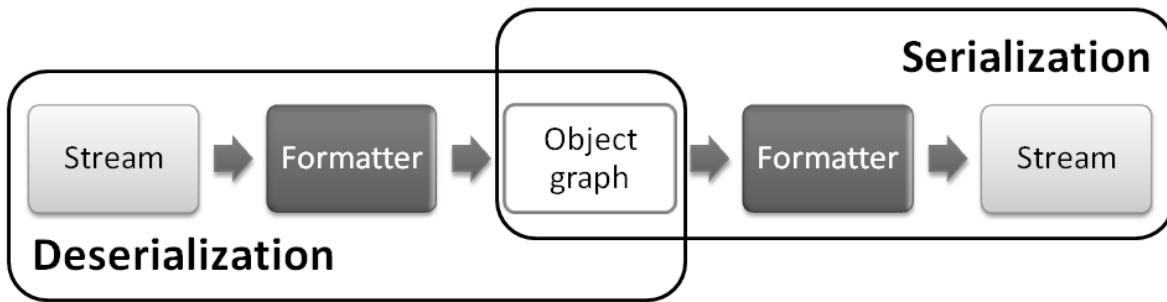
Having said that, it's evident that most real-world applications use serialization and deserialization in various forms and flavors without even realizing it. In this module, we will focus on understanding how .NET serialization works and what other alternatives for serialization exist in the .NET world.

MCT USE ONLY. STUDENT USE PROHIBITED

4-6 | Module 04 - Serialization

.NET Serialization

- All primitive types are serializable.
- Mark custom types with [Serializable]



.NET serialization, also termed runtime serialization, is a fundamental feature of the .NET framework that is not associated with any particular application framework on top of .NET (such as WCF, WPF or any other). Runtime serialization assumes that all primitive types are serializable (including the fundamental value types, strings, and arrays of the above), and that custom user types are marked as serializable on an opt-in basis.

Discussion: Is opt-in serialization reasonable? Why not serialize every object? (This is the approach, by the way, taken by Data Contract Serialization in WCF 3.5 SP1 – any plain object can be serialized by default.)

Marking types for serialization is done by decorating them with the [Serializable] attribute, which is an attribute queried by the framework at runtime. (In fact, this attribute is so important that it has its own metadata entry, unlike most other attributes.)

The serialization process consists of taking an object graph (which might contain one or more objects, including cyclic references between them) and using a *formatter* to output the object graph to a stream. There are no limitations on the stream, the object graph or the serialization format, and the .NET framework includes two built-in formatters which we will see later.

The deserialization process consists of taking a stream and using a formatter to construct an object graph from the stream. Objects that had references before serialization will retain these references after serialization – the reconstruction is full and exactly reflects the state of the objects before serialization. If the stream is invalid or if the formatter produces an unexpected object, runtime errors might occur and must be taken care of.

Discussion: Do we always want to serialize everything? What are the scenarios for not serializing part of the object, or dynamically reconstructing parts of an object?

Serializable User Class

```
1 [Serializable]
2 class User
3 {
4     private string _name;
5     private string _password;
6     private DateTime _loginTime;
7     private int _reputation;
8     public User(string name, string password)
9     {
10         _name = name;
11         _password = password;
12         _loginTime = DateTime.Now;
13         _reputation = ReputationDB.Reputation(name);
14     }
15 }
16 }
```

CODE

This example shows a simple class called User which stores some properties about a logged-on user, including the user name, password, login time and reputation points. The class is marked with the [Serializable] attribute, making its contents automatically serializable. All fields of the User class are serializable individually, making the entire object serializable.

Note that this example already demonstrates some of the problems with default serialization: The user's password should not be serialized for security reasons, and the user's login time and reputation should not be serialized because they are volatile and subject to change. You will see how this can be dealt with in the rest of this module.

MCT USE ONLY. STUDENT USE PROHIBITED

BinaryFormatter

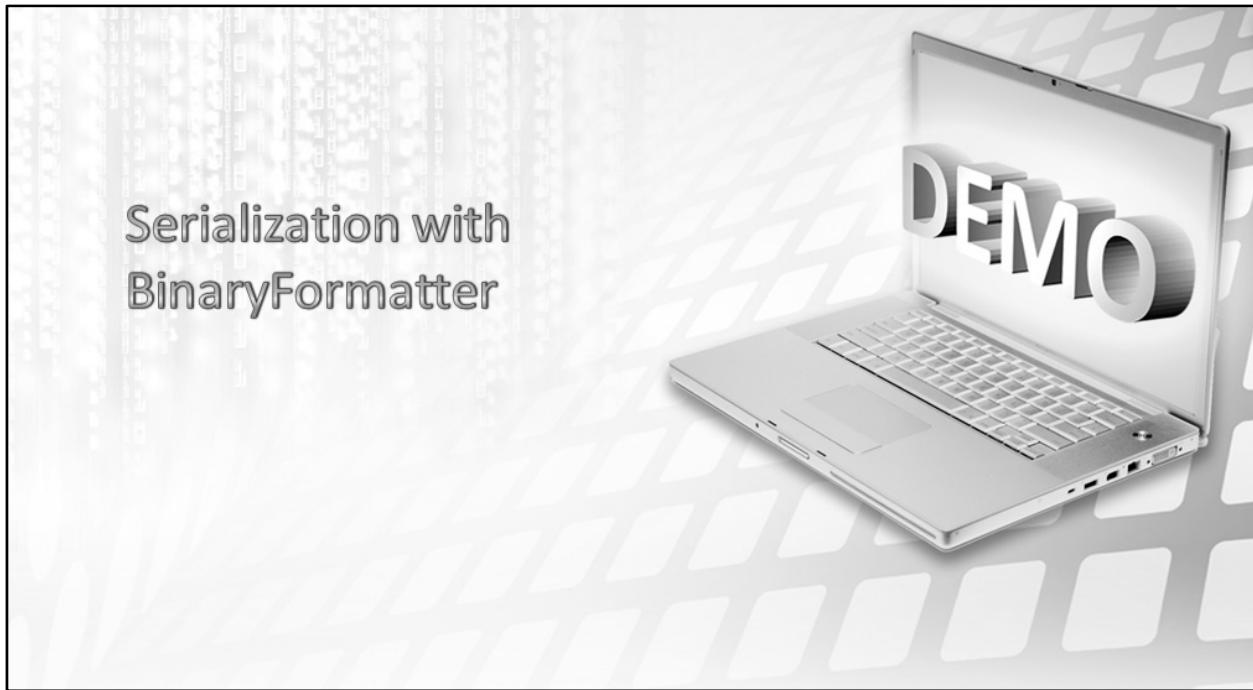
```
1 User joe = new User("Joe", "123456");
2 BinaryFormatter formatter = new BinaryFormatter();
3 FileStream file = File.Create("joe.serialized");
4 formatter.Serialize(file, joe);
5 file.Close();
6
7 file = File.Open("joe.serialized", FileMode.Open);
8 joe = (User)formatter.Deserialize(file);
9 file.Close();
```

- Implements Formatter
- Serialize and deserialize:

Serialization and deserialization relies on *formatters*, which are the entity that constructs an object graph from a stream of bytes and vice versa. A formatter in .NET is represented by the *Formatter* interface, which contains a *Serialize* and *Deserialize* method pair for performing the fundamental operations. The *BinaryFormatter* class performs binary serialization, producing a fairly compact representation of an object in a binary stream of data. Usually the serialization results from binary serialization are unreadable by humans. However, this does not mean that binary serialization is encrypted or inherently secure!

This example demonstrates serialization and deserialization of a User instance (see the previous slide for a description of the User class) to a file called “joe.serialized”. The serialized file contents for all examples in this chapter are available as part of the demo code, under the **Module04_Serialization** solution in the SampleCode directory.

Lines 1-5 entail the serialization of a User instance. A BinaryFormatter instance is created, a FileStream is opened and the Serialize method performs the serialization. In lines 7-9, another FileStream is used to read the serialization results into a User instance.



Serialization with BinaryFormatter

See the **SerializationTypes** project in the **Module04_Serialization** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you'll see how an object is serialized to binary file, along with all its private fields. You will become aware of two problems with this approach:

- Sensitive information might leak into the serialized file. Even though the file is not easily readable by humans, opening it in the Visual Studio editor reveals the user's password without any effort.
- Dynamic information might not be reflected in the deserialized instance. In this example, the user's reputation is not refreshed when the instance is deserialized. The login time is not refreshed either, providing false information to the system.

In the subsequent demos, you will see how these security and correctness issues can be mitigated by customizing the serialization process.

MCT USE ONLY. STUDENT USE PROHIBITED

4-10 | Module 04 - Serialization

Controlling Serialization

- ① Some fields can't be serialized
 - ⌚ Volatile data
 - ⌚ Sensitive data
- ② Use [NonSerialized]

```
1 [Serializable]
2 class User2
3 {
4     private string _name;
5     [NonSerialized] private string _password;
6     private DateTime _loginTime;
7     private int _reputation;
```

CODE

As we've seen in the previous demo, some object data shouldn't be serialized because of volatility and sensitivity reasons. The [NonSerialized] attribute is the first way of controlling serialization by indicating that a specific field of a type should not be serialized at all.

In this example, line 5 shows how the `_password` private field is marked with the [NonSerialized] attribute to address the sensitivity concerns around this information. When an instance of the `User2` class is serialized, the resulting stream does not contain this data field. When the stream is used for deserialization, the resulting instance will have a null string instead of the password.

However, we have still not addressed the need to dynamically repopulate the reputation and login time information when the instance is deserialized. The [NonSerialized] attribute can't help us out here – we are in need of a stronger mechanism for controlling serialization.

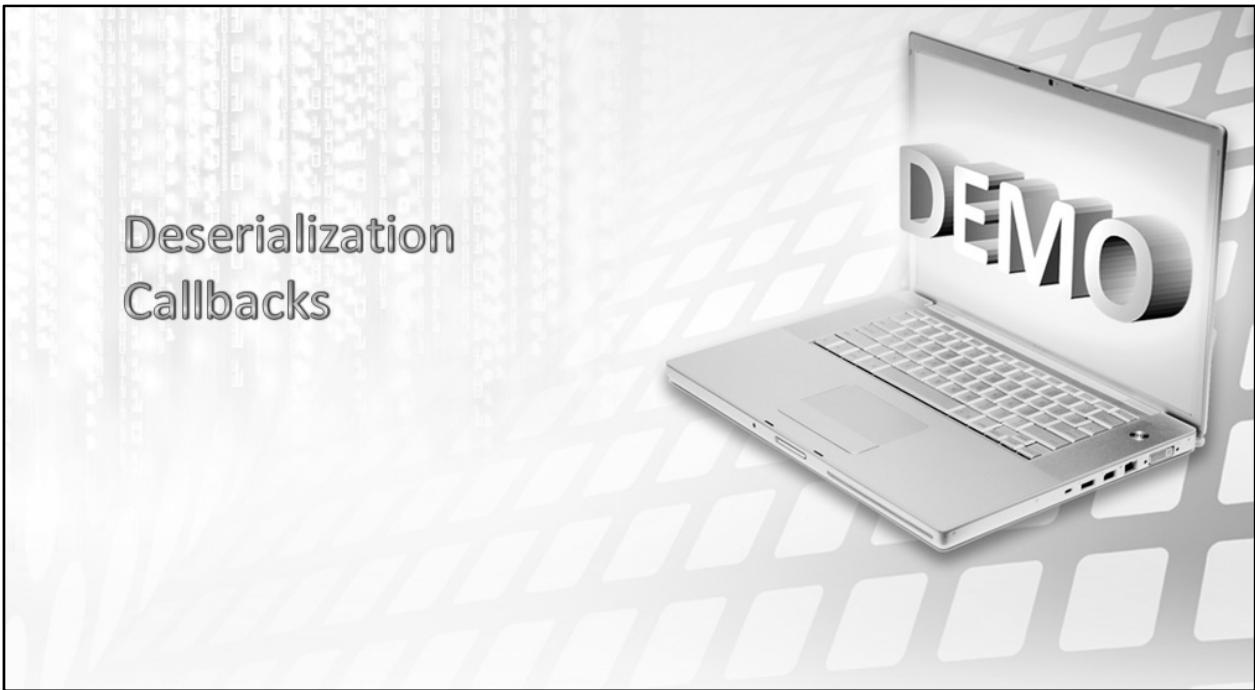
Serialization Callbacks

- [OnSerializing]
- [OnSerialized]
- [OnDeserializing]
- [OnDeserialized]

The .NET serialization and deserialization mechanism allows for customization by providing a set of four hooks (callbacks) into the serialization and deserialization process. These callbacks are implemented as methods on the type being serialized, and are decorated with one of the four attributes that appear on the slide: [OnSerializing], [OnSerialized], [OnDeserializing] and [OnDeserialized]. Each attribute indicates to the runtime that the method it decorates should be called when the corresponding operation is performed:

- [OnSerializing] decorates a method that is invoked before serialization begins. This is useful for concealing information or otherwise affecting the serialization process.
- [OnSerialized] decorates a method that is invoked after serialization has completed.
- [OnDeserializing] decorates a method that is invoked before deserialization begins.
- [OnDeserialized] decorates a method that is invoked after deserialization has completed. This is useful for reconstructing dynamic information, in our case the volatile reputation and login time fields.

All callback methods should typically be private instance methods, and accept a single parameter of type `StreamingContext`. The `StreamingContext` parameter allows you to determine the context of the serialization operation, e.g. if serialization is performed in order to pass an object through a network boundary or to store it into a file. Although unlikely, this information might affect your decision on the serialization format of the object's data.



Deserialization Callbacks

See the **SerializationTypes** project in the **Module04_Serialization** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how a deserialization callback (using the [OnDeserialized] attribute) placed on the User class we have seen previously allows for dynamic recovery of the reputation and login time fields after deserialization occurs. The deserialized instance has its login time and reputation points data reconstructed dynamically after deserialization, making the whole process completely transparent to the client of the class.

Custom Serialization

- Implement `ISerializable`
- Add a protected constructor taking `SerializationInfo`, `StreamingContext`

What we have seen so far is how the `[NonSerialized]` attribute and the serialization and deserialization callbacks can be used to customize and control the serialization and deserialization process. However, what brings us closest to complete control over serialization and deserialization format is the `ISerializable` interface, which contains a single method called `GetObjectData` and taking a `SerializationInfo` and `StreamingContext` parameters. By convention, to implement custom deserialization, a class will also declare a private or protected constructor taking the same two parameters. The `GetObjectData` method and this special constructor will be invoked by the runtime during serialization and deserialization, respectively.

The power of the custom serialization enabled by `ISerializable` is that the `SerializationInfo` instance used by the serialization and deserialization process can be instructed to contain exactly the information desired for the custom serialization process. The serialization information is maintained as key-value pairs, and there does not have to be a one-to-one correspondence between the type's data members and the serialized key-value pairs. Compression and encryption are just two examples, but other scenarios for custom control over serialization are widely prevalent.

4-14 | Module 04 - Serialization

Custom Serialization Example

```
1 [Serializable]
2 class User4 : ISerializable
3 {
4     void ISerializable.GetObjectData(
5         SerializationInfo info, StreamingContext context)
6     {
7         info.AddValue("Name", _name);
8         info.AddValue("Password", Encrypt(_password));
9     }
10    protected User4(
11        SerializationInfo info, StreamingContext context)
12    {
13        _name = info.GetString("Name");
14        _password = Decrypt(info.GetString("Password"));
15        InitLoginTimeAndReputation();
16    }
```

A large, stylized, three-dimensional white text logo with a black shadow effect, spelling the word "CODE".

This example demonstrates the use of custom serialization to encrypt the user's password when serializing it and decrypt it when deserializing it. The Encrypt and Decrypt functions are assumed to perform this process (and in the actual demo project they are implemented using the insecure XOR-based encryption algorithm).

The GetObjectData is explicitly implemented (line 4) because there is no reason to expose the fact that a User object is ISerializable – this is an interface that is generally only useful for the serialization framework, and should not be directly exposed to client code.

Additionally, line 15 demonstrates how custom deserialization can be used to dynamically generate volatile data fields which were not serialized with the object. Note that the class has the same fields as before, but they have been omitted for presentation brevity.

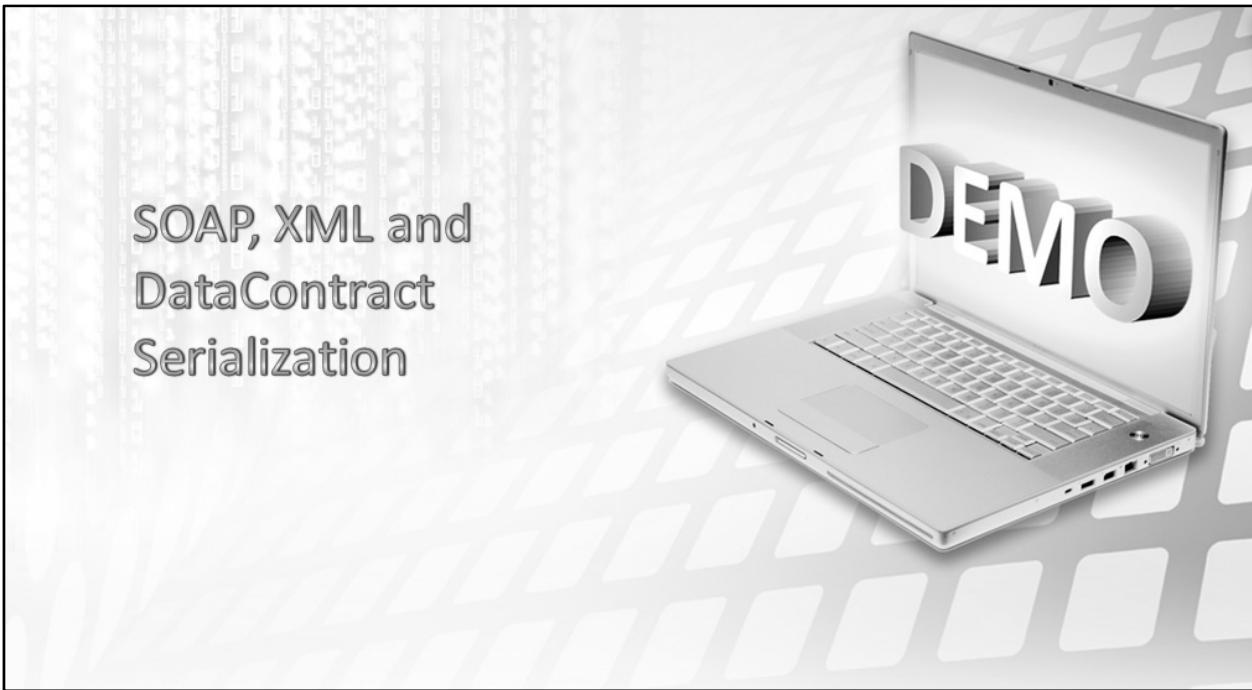
Serialization Alternatives

- SoapFormatter – deprecated
- XmlSerializer – public data, no cycles
- DataContractSerializer – WCF

Alternatives for the .NET runtime binary serialization mechanism include:

- SoapFormatter, which is a deprecated formatter implementation serializing object graphs to SOAP envelopes. (This formatter does not support generics and therefore is considered deprecated and its use is generally discouraged.)
- XmlSerializer, which is a serializer targeting acyclic object graphs with public get/set properties.
- DataContractSerializer, which is the WCF (Windows Communication Foundation) serializer of choice, capable of serializing cyclic object graphs to a variety of formats, including SOAP-like XML.

While there is a place for using each serialization framework, the most commonly used framework today is the .NET runtime serialization mechanism. It is used under the covers by .NET features such as Remoting (and for communication between application domains), and will remain part of the framework in the future. Other serialization alternatives have appealing scenarios of their own, which are somewhat demonstrated in the demo. The other serialization mechanisms are not the primary focus of this chapter, and therefore you are encouraged to learn more about them independently.



SOAP, XML and DataContract Serialization

See the **OtherSerializationMechanisms** project in the **Module04_Serialization** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how the SoapFormatter, the XmlSerializer and the DataContractSerializer classes can be used to serialize and deserialize object graphs. The pros and cons of each serializer are briefly mentioned in the demo, and examples of the serialization output are part of the demo project.

A somewhat more comprehensive example is provided by the **XmlConfigurationFramework** project in the same solution, which demonstrates how a simple alternative to the .NET configuration framework can be built using the simplest features of XML serialization.

Lab: Serialization Framework



Serialization Framework Lab:

Use the **SerializationFramework_Starter** solution under the **Exercises\Module04_Serialization** folder as a starter solution for implementing a serialization framework with support for encryption, compression and dynamic deserialization of an object's fields.

Using the supplied encryption, decryption, compression and decompression code, you will read the serialization traits provided using the [Serialization] and [DeserializationCallback] attributes and implement a serialization framework which performs custom seamless serialization.

You can find the solution for this lab in the **SerializationFramework_Solution** solution under the **ExerciseSolutions\Module04_Serialization** folder.

MCT USE ONLY. STUDENT USE PROHIBITED

Summary

- Motivation for serialization
- Marking a type for serialization
- BinaryFormatter
- Controlling serialization
- Custom serialization
- Overview of XML serialization
- Overview ofDataContract serialization
- Lab





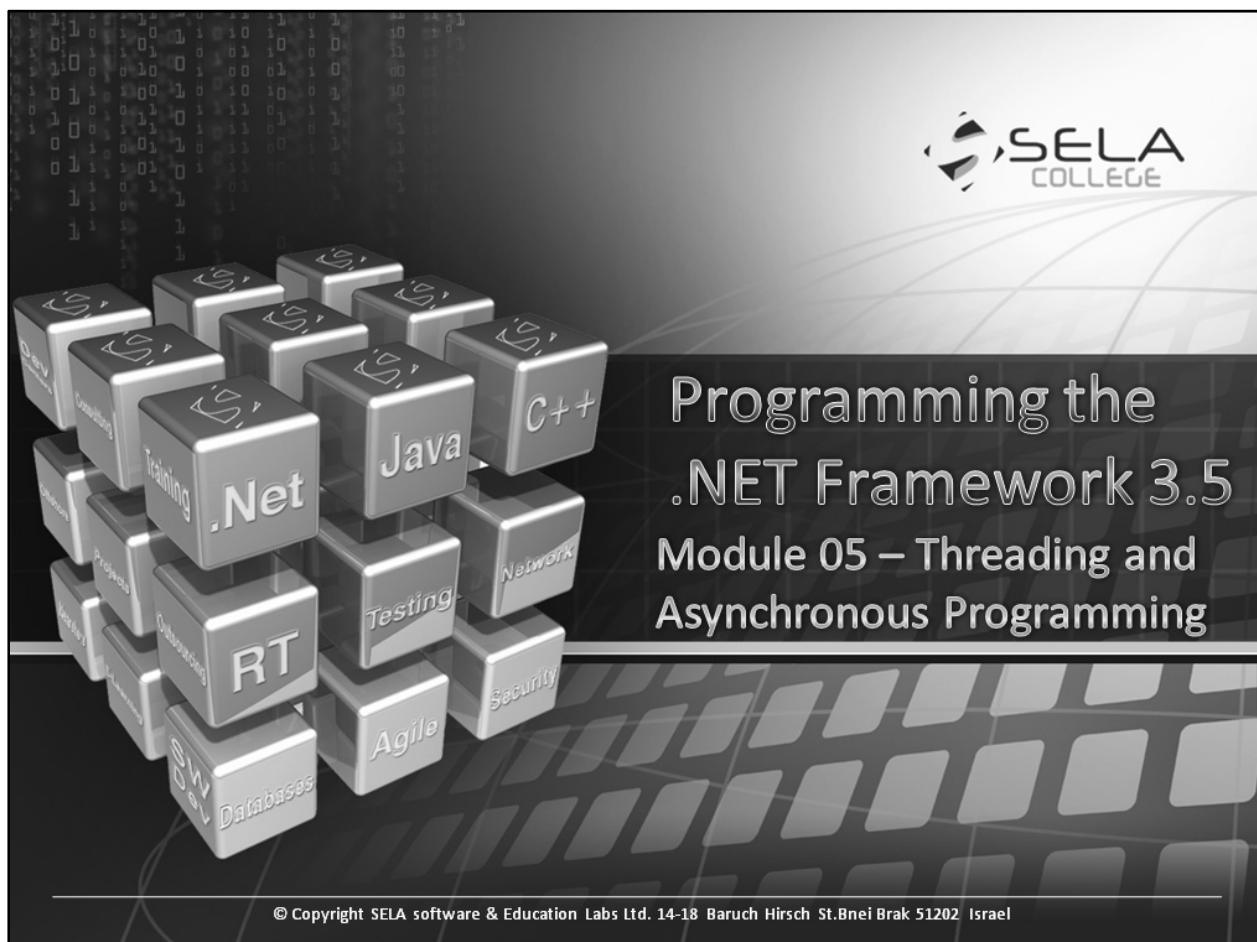
Module 05 - Threading and Asynchronous Programming

Contents:

In This Chapter.....	4
Multi-Tasking and Multi-Processing	5
Processes and Threads	6
One Program, Multiple Threads	7
Why Threads?	8
Why Not Threads?	9
Scheduling at a Glance	10
Amdahl's Law.....	11
Asynchronous Programming Model.....	12
APM and Files.....	13
APM and Threads	16
BeginInvoke and EndInvoke	17
Various Ways to End.....	18
Various Ways to End (contd.)	19
Maintain State With a Callback	20
Maintain State with AsyncState	21
BackgroundWorker	23
Lab: APM in the WinForms UI.....	25
Queuing Work for Execution	26
ThreadPool.QueueUserWorkItem.....	27
Manual Threading	29
Thread Class	30
When Does It End?	31

MCT USE ONLY. STUDENT USE PROHIBITED

Abort vs. Interrupt.....	32
Inter-Thread Communication	33
Shared Data → Synchronization.....	35
Shared Data Races.....	37
Busy Synchronization.....	38
Critical Sections	39
Monitor and Lock.....	40
Deadlocks.....	42
Deadlocks in Code	43
Pulse, PulseAll, Wait, WaitAll.....	45
WaitHandle Synchronization	46
Using Events for Synchronization	47
Parallel Extensions for .NET	49
Lab: Thread-Safe Resource Parallelizing Work.....	50
Summary	51



MCT USE ONLY. STUDENT USE PROHIBITED

5-4 Module 05 - Threading and Asynchronous Programming

In This Chapter

- ⌚ Multi-tasking, processes, threads, asynchrony, scheduling
- ⌚ Asynchronous programming model (APM)
- ⌚ Thread pool
- ⌚ Manual threading
- ⌚ Synchronization
- ⌚ Overview of Parallel Extensions for .NET
- ⌚ Lab

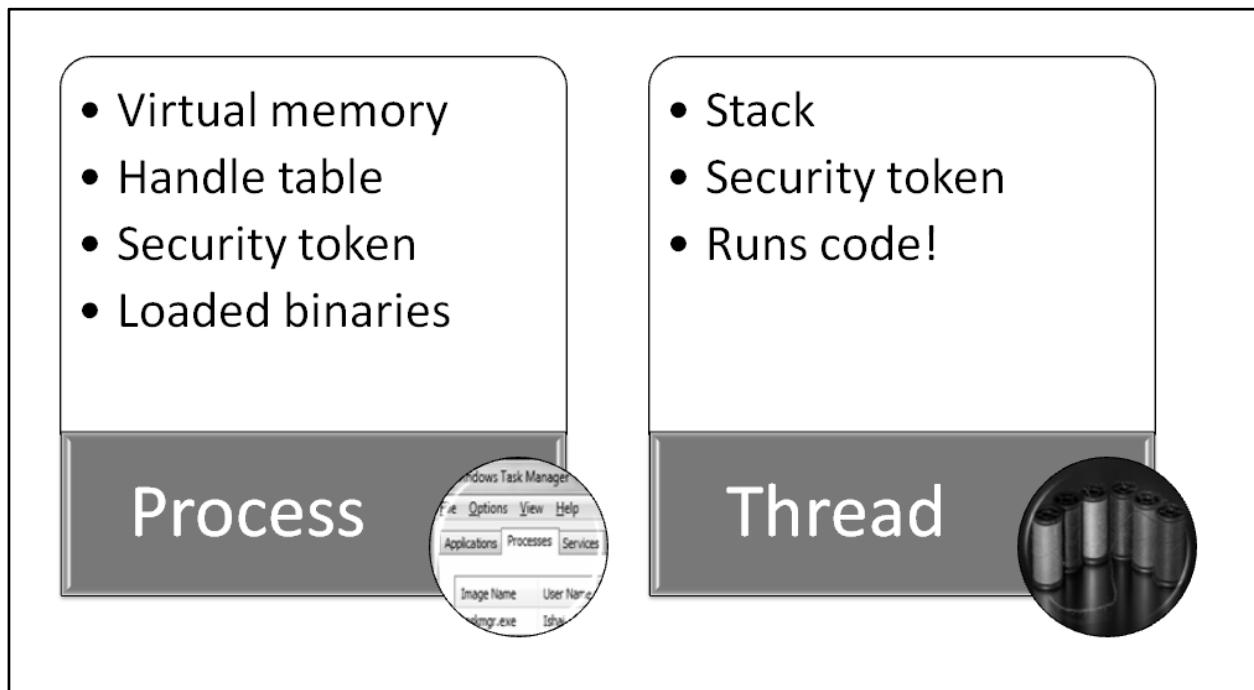


Multi-Tasking and Multi-Processing

- Executing multiple tasks at once
- Executing tasks on multiple processors

Before we begin with the intricacies of developing multi-threaded applications using the .NET Framework, it's important to have some common context and terminology fleshed out. To begin with, multi-tasking is about executing multiple tasks at once on the same machine. The "at once" clause might all be about maintaining an illusion of simultaneous execution, leaving the user none the wiser. For example, even though I have only one processor on my Asus EEE PC netbook, I am still capable of executing multiple tasks at once – I can browse the web, write an email and read a PDF book without having to finish one activity and close the application before beginning another.

On the other hand, the term multi-processing is usually about executing multiple tasks at once on multiple processors. It means that the operating system supports the notion of multiple tasks as well as the ability to distribute these tasks across a set of more than one physical processor. Almost all operating systems today are multi-tasking as well as multi-processing operating systems. Specifically, Windows can be characterized as a *preemptive multi-tasking symmetric multi-processing operating system*.

5-6**Module 05 - Threading and Asynchronous Programming****Processes and Threads**

The Windows operating system manages tasks and multi-processing using the notions of processes and threads. A process is an execution environment in which threads execute (the process itself does not execute code – it relies on threads to do so). A process is bundled together with a virtual address space (4GB on 32-bit systems, of which typically 2GB are accessible), with a handle table to access kernel objects (including synchronization objects, covered in this module), a security token and a set of loaded binaries that can be executed. A thread represents a context of execution within the process' environment, and so all threads within the process share the same virtual address space, the same handle table and the same default security token (although the security token can be customized). Threads have a stack which is used to declare local variables and invoke functions, and is the entity that actually executes code.

Every process is associated with at least one thread of execution when it is created. This is the thread that executes the entry point of the process' executable file (i.e., the “main” method).

One Program, Multiple Threads

- Threads can execute at different points of the same code
- Threads can execute **simultaneously**

Multiple threads of execution can be present in one process at the same time. Threads can execute the same code or different portions of the code. It is possible for the same data to be accessed for reading or for writing from more than one thread of execution, just as it is possible for the same code to be executing in one or more thread of execution.

Due to the operating system guarantee of multi-tasking and multi-processing, threads execute simultaneously. Although this provides for responsiveness and performance improvements (as we will see shortly), it can also be a source of very difficult errors.

5-8 | Module 05 - Threading and Asynchronous Programming

Why Threads?

- Deferred background work
- Parallelization of work
- Program structure

Threads are often used to defer background work to a separate execution context. This allows one part of the application to remain responsive while another part of the application (another thread) performs a background calculation. This is very typical in UI applications, where the user interface must remain responsive even if background work (such as printing, saving or paginating a document) has to occur in the background.

Threads are also used to parallelize work, because a single thread can execute on a single processor, but multiple threads can take advantage of multiple processors at the same time. Tasks that are often categorized as “embarrassingly parallel” benefit the most from parallelization. For example, calculating prime numbers or constructing complicated meteorological models is a task that is easily split to multiple threads of execution.

Parallelization of such tasks often provides speedup that is linear in the number of processors used.

Finally, threads are also sometimes used to structure the program in a way that better models the actual problem domain. For example, if the problem domain involves asynchronous interaction between entities, this interaction is often best modeled by the use of threads and a message queue that one thread uses for enqueueing work and the other uses for dequeuing it.

Why Not Threads?

- Corruption of shared data
- Contention for shared data
- Thread affinitized resources
- Program structure

Although we have just seen why threads can be very useful, they also have significant disadvantages, most of them stemming from the fact that multi-threaded applications are more prone to certain categories of difficult errors, which are very hard to diagnose and fix.

Threads are often using the same shared data, and if the same shared data is modified by multiple threads of execution it can easily become corrupted. For example, while one thread is updating a certain field in a database, another thread might be updating a related field. The two modifications can easily be performed out of sync if they occur truly simultaneously. Even if access to shared data is synchronized (so that only one thread accesses it at every given time), this very synchronization is the source of bottle-necks in heavily parallelized applications. Whenever data must be protected from concurrent access, it becomes a bottleneck for scaling the application across more and more processors.

Additionally, some resources in applications require thread affinity – i.e., only one thread (often a specific thread) is allowed access to them at a given time. For example, Windows UI elements have to be accessed from the same thread that was used to create them. Failure to comply with this requirement leads to very difficult programming errors.

Finally, the same argument about program structure can be used in the inverse: If threads are a good model of certain domain elements, then often they are also a very bad model of it.

Interactions in the real world are often synchronous and serialized, and modeling them for concurrent execution introduces bugs and design flaws which are often very hard to reconcile.

5-10 | Module 05 - Threading and Asynchronous Programming

Scheduling at a Glance

- A thread has a priority
- The single highest-priority ready thread always runs
- SMP scheduling is very hard
- Starvation
- Synchronization convoys

It's important to know that the operating system schedules threads based on their priority. Although most threads in the system have the same priority (which is determined by a combination of the base priority of thread's process with the relative priority of a thread within that process), some threads such as operating system threads opt to modify their priority. Although this is outside the scope of this course (consider the Windows Internals or Windows Concurrent Programming courses for more information), modifying the priority of application threads should be performed with great care, as it has the potential of hindering system activity or even adversely affecting the same application that uses priority modifications.

Windows guarantees that the single highest-priority thread that is not currently blocked (i.e. ready) always runs. On a multi-processor system, contrary to what can be expected, Windows does not guarantee that for N processors, the N highest-priority ready threads always run. Scheduling on a multi-processor system is extremely complicated, and is outside of the scope of this module.

Finally, bear in mind that priority scheduling and scheduling in general are subject to theoretical and practical problems such as starvation, priority inversion and synchronization convoys. These problems are also outside the scope of this course.

Amdahl's Law

- If P is the proportion of code that can be parallelized, then the maximum possible speedup (with N processors) is:

$$S = \lim_{N \rightarrow \infty} \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{1-P}$$

- For P=90%, S=10
- For P=50%, S=2

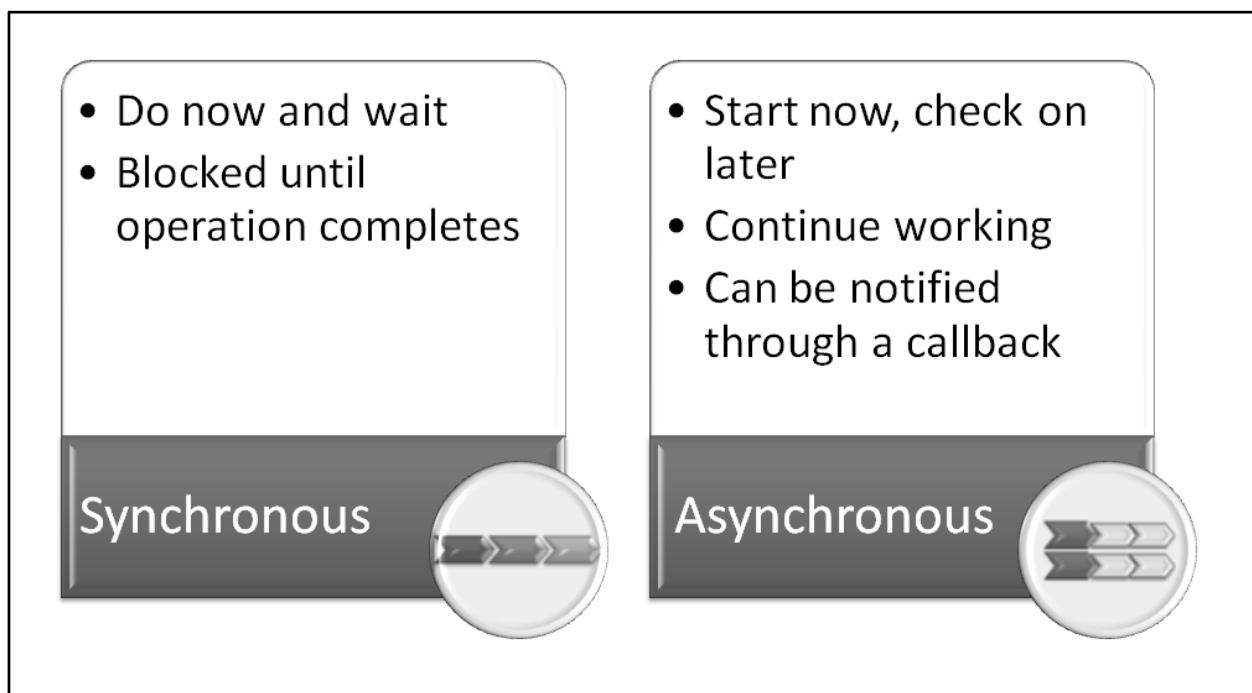
Amdahl's law, also known as "The Law of Diminishing Returns", provides a formula to determine the maximum possible speedup (S) of an application using N processors, as a function of the amount of code that can be parallelized.

Clearly, an application that is fully parallelizable can be sped up to infinity. However, any application that is not fully parallelizable will reach a point of diminishing returns. This law is also known as the law of diminishing returns because at some point (where P < 1), adding processors (N) is less and less useful (logarithmic convergence).

This might be a surprising result, but it's extremely important when designing a system that has to scale to more than 2-4 processors. A scalability bottleneck of 10% is almost invisible when scaling up to 2 or 4 processors (if X is the single processor runtime, then the 2 processor runtime will likely be close to X/2), but as more and more processors are added, the scalability bottleneck can be sensed more clearly.

5-12 | Module 05 - Threading and Asynchronous Programming

Asynchronous Programming Model



The .NET Framework provides an asynchronous programming model which is arguably the easiest way in .NET to defer work to another thread. While it is not always suitable for parallelization tasks, it is perfectly suitable for performing background work in another thread. A synchronous programming model involves a call to a certain operation, and a wait until that operation completes. In between, the waiting thread of execution does not do anything but wait for the operation to complete. (Alternatively, it might be executing the operation – but no other work can proceed in the meantime.)

Alternatively, an asynchronous programming model involves starting an activity on another (background) thread of execution, proceeding to perform other work and later checkpointing with the background activity through the use of a callback (push) or an explicit question or wait (poll).

Asynchronous execution is inherently harder to understand, but it is often used to provide UI responsiveness (an example we have already seen) and sometimes models the real domain more closely than synchronous execution (for example, when asking another person a question, we are free to continue thinking while waiting for an answer – this is an inherently asynchronous model).

APM and Files

```
1 while (true) {  
2     ar1 = reader.BeginRead(buf1, 0, 8192, null, null);  
3     while (!ar1.IsCompleted) ...  
4     if (ar2 != null)  
5         while (!ar2.IsCompleted) ...  
6     if ((read = reader.EndRead(ar1)) == 0)  
7         break; //No more data to read  
8     if (ar2 != null)  
9         writer.EndWrite(ar2);  
10    Array.Copy(buf1, buf2, read);  
11    ar2 = writer.BeginWrite(buf2, 0, read, null, null);  
12 }
```

Specific example:

- BeginRead
- EndRead
- BeginWrite
- EndWrite

The asynchronous programming model (APM) is implemented in several .NET types out of the box. The more general support is available with every delegate, and will be our next subject of exploration.

For now, let's take a look at the FileStream API. The FileStream class, which is used to read from and write to files, has Read and Write functions as well as the asynchronous versions:

BeginRead, EndRead and BeginWrite, EndWrite pairs. After calling the BeginRead operation, for example, the calling code does not block until the entire read request is satisfied – it is free to proceed with other tasks and checkpoint with the read operation by calling EndRead or by using the IAsyncResult interface returned from the BeginRead call.

The IAsyncResult interface contains, among other members, the IsCompleted boolean property which can be used to poll the completion result of the read operation initiated with BeginRead. To obtain the actual return value of the Read method, which is the number of bytes transferred, the EndRead method must be used, and the IAsyncResult object returned from the BeginRead call must be passed to EndRead as a parameter (this is required because multiple BeginRead operations can be issued before EndRead is called, and there must be a way to correlate the EndRead call to a preceding BeginRead call).

In this example, reads and writes are interleaved, producing a file copy operation that is likely faster to complete than its synchronous counterpart. Within the loop, the code first initiates a BeginRead operation for 8KB of data, and stores the IAsyncResult returned in a local variable. Until the read completes, the code stalls. Next, EndRead is called and if there is no more data in the file, the while loop is aborted. If there still was some data, the data is copied to another buffer and the BeginWrite operation is issued asynchronously, storing the IAsyncResult in

5-14 | Module 05 - Threading and Asynchronous Programming

another local variable. The loop then repeats, and this time after issuing the asynchronous read it will checkpoint for both the read and write to complete. However, at that point in time, the read and write will actually proceed in unison, taking advantage of the disk's ability to perform a read and a write at the same time. (Note that even if the target device is not capable of actually interleaving reads and writes, it might as well be the case that the reads are coming from a network device and the writes target an SSD flash drive – the I/O paths to these devices are different and therefore there is a very good reason to interleave them.)

APM and Files



See the **APM** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how asynchronous reads and writes are interleaved to achieve better performance when copying files by taking advantage of both the read and write I/O paths at the same time.

5-16 | Module 05 - Threading and Asynchronous Programming

APM and Threads

- Threads provide the execution fabric
- .NET delegates provide the asynchrony
- `BeginInvoke`, `EndInvoke`
- `Invoke`

The previous example and demo showed how the `FileStream` class, which provides explicit asynchronous operations, can be used to orchestrate asynchronous read and write work. However, the general question remains – how does the APM (asynchronous programming model) allow us to execute asynchronous operations on *any* target, not just a target (like `FileStream`) which was designed to support asynchronous operations?

The answer lies in threads of executions and .NET delegates, along with some compiler magic. Threads provide the underlying execution fabric which can be used to perform an asynchronous operation. While the calling thread proceeds to mind its own business, another thread in the system (managed by the thread pool, which we will see later in this module) executes the asynchronous request. The two threads provide the means for asynchronous execution.

.NET delegates, on the other hand, provide the asynchronous semantics for delegate invocation. A .NET delegate can be invoked using the () function call syntax or using the `Invoke` method, but it is also automatically supplied by the compiler with `BeginInvoke` and `EndInvoke` methods, which mirror the way `BeginRead` and `EndRead` worked for the `FileStream` class in the previous example. By using the `BeginInvoke/EndInvoke` support with any delegate type, programmers are free to use the APM for any operation.

BeginInvoke and EndInvoke

```
1 var asyncSieve = new PrimeNumberCalculator(...);
2 Func<PrimeNumberCalculation, int> asyncSieveCalc =
3     asyncSieve.Calculate;
4 IAsyncResult ar1 = asyncSieveCalc.BeginInvoke(
5     PrimeNumberCalculation.Sieve, null, null);
6 var asyncStandard = new PrimeNumberCalculator(...);
7 Func<PrimeNumberCalculation, int> asyncStandardCalc =
8     asyncStandard.Calculate;
9 IAsyncResult ar2 = asyncStandardCalc.BeginInvoke(
10    PrimeNumberCalculation.Sieve, null, null);
11 while (!(ar1.IsCompleted && ar2.IsCompleted))
12     Thread.Sleep(100);
13 Console.WriteLine("{0} primes using the sieve",
14     asyncSieveCalc.EndInvoke(ar1));
15 Console.WriteLine("{0} primes using standard method",
16     asyncStandardCalc.EndInvoke(ar2));
```

CODE

This slide demonstrates an asynchronous invocation of a prime number calculation. Two prime number calculations are launched in parallel – one that uses the sieve method and another that uses the standard method of dividing all numbers up to the square root of the desired number.

The parallelism is achieved by creating two delegates of type

`Func<PrimeNumberCalculation,int>` which are then invoked asynchronously using the `BeginInvoke` method.

The code then waits (line 11) for both calculations to complete, and then displays the execution results by using the `EndInvoke` method on the two delegates. Note that because we used `BeginInvoke` on both delegates, the calculations actually ran in parallel – if you examine the CPU utilization on a dual-core machine, you will see both CPUs (100%) utilized to their full capacity, because prime number calculation is mainly a CPU-intensive, CPU-bound task.

MCT USE ONLY. STUDENT USE PROHIBITED

5-18 | Module 05 - Threading and Asynchronous Programming

Various Ways to End

- EndInvoke
- Poll IsCompleted
- Wait for AsyncWaitHandle
- Register callback

So far we have looked at two ways to checkpoint with the asynchronous execution of an operation – using the EndInvoke method, which waits for completion if the operation was not finished yet, and the IsCompleted boolean property of the IAsyncResult interface which allows the calling code to periodically poll for completion without waiting.

There are two other alternatives for checkpointing with the asynchronous operation. The IAsyncResult interface returned by BeginInvoke contains an AsyncWaitHandle property which is of type WaitHandle. It can be used to explicitly wait (with a timeout) until the asynchronous operation completes. We will see how the WaitHandle class can be used later in this module. Another option is to provide the BeginInvoke operation with a callback (an AsyncCallback delegate) that will be invoked when the operation completes. The result of the operation can then be obtained by calling BeginInvoke with the same IAsyncResult instance returned from BeginInvoke.

Various Ways to End (contd.)

```
1 var calc = new PrimeNumberCalculator(5, 100000);
2 Func<PrimeNumberCalculation, int> invoker =
3     calc.Calculate;
4
5 //Poll for IsCompleted property:
6 IAsyncResult ar = invoker.BeginInvoke(
7     PrimeNumberCalculation.Sieve, null, null);
8 while (!ar.IsCompleted)
9     Thread.Sleep(100);
10 int result = invoker.EndInvoke(ar);
11
12 //Wait for the wait handle:
13 ar = invoker.BeginInvoke(
14     PrimeNumberCalculation.Sieve, null, null);
15 ar.AsyncWaitHandle.WaitOne();
16 result = invoker.EndInvoke(ar);
```

CODE

This example shows how the `IsCompleted` property and the `AsyncWaitHandle` property can be used to wait for the operation to complete. `IsCompleted` is a trivial boolean property which returns immediately, while `AsyncWaitHandle` can be used to wait with a timeout (not shown in this demo) by using one of the overloads of the `WaitOne` method.

MCT USE ONLY. STUDENT USE PROHIBITED

5-20 | Module 05 - Threading and Asynchronous Programming

Maintain State With a Callback

- ① Easy with closures
 - ⌚ Or could use AsyncState property

```
1 //Use callback
2 ar = invoker.BeginInvoke(
3     PrimeNumberCalculation.Sieve, delegate
4     {
5         result = invoker.EndInvoke(ar);
6     }, null);
```

CODE

Another option is to use a callback, and in the C# 2.0/3.0 era where anonymous methods and lambdas provide semantic closures – it's fairly easy to coordinate the asynchronous operation and its completion work. Specifically, in this example the BeginInvoke operation provides a delegate (with its anonymous method body in lines 4-6) which uses the EndInvoke operation on the IAsyncResult variable *ar* that is returned from the BeginInvoke call.

Note that due to a limitation of the C# compiler, it's impossible to declare and initialize the IAsyncResult *ar* variable in the same statement, because it is then used as part of the same statement (in line 5, when calling the EndInvoke method). Therefore, first declare the variable and initialize it to null (not shown in this example) and then assign to it the result of the BeginInvoke method.

Maintain State with AsyncState

```
1 //Use callback without closures:  
2 Printer printer = new Printer();  
3 ar = invoker.BeginInvoke(PrimeNumberCalculation.Sieve,  
4                         new AsyncCallback(CalculationEnded), printer);  
5  
6 private static void CalculationEnded(IAsyncResult ar)  
7 {  
8     //We need to retrieve the delegate and the state:  
9     AsyncResult realAR = (AsyncResult)ar;  
10    Printer printer = (Printer)ar.AsyncState;  
11    var invoker =  
12    (Func<PrimeNumberCalculation,int>)realAR.AsyncDelegate;  
13    //End the operation and print the result:  
14    int result = invoker.EndInvoke(ar);  
15    printer.Print(result);  
16 }
```

CODE

If for some reason you're not using an anonymous method or a lambda expression to specify the callback, then you have to work slightly harder to retrieve the original delegate instance and the IAsyncResult instance returned from BeginInvoke so that you can call EndInvoke successfully. In this example, line 3 uses BeginInvoke and provides an AsyncCallback delegate which points to the CalculationEnded method. Additionally, this time we're also taking advantage of the last parameter to BeginInvoke – an object parameter which can be used to pass state along to the callback method that is invoked when the operation completes. In this case, we're passing an instance of the Printer class as a parameter.

In the callback itself, we need to retrieve the delegate and the IAsyncResult instance. The IAsyncResult instance is passed as a parameter to the AsyncCallback delegate we registered, and from it we can obtain (through the AsyncState property) the Printer object that we passed along. To obtain the delegate itself, we resort to casting the IAsyncResult parameter to the underlying AsyncResult implementing class, and from that class we can obtain the actual delegate by using the AsyncDelegate property (lines 9 and 11). Finally, in line 14 we use the EndInvoke method to obtain the calculation result, completing the asynchronous invocation.



See the **APM** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how various forms of checkpointing with an asynchronous operation can be used in the calling code, which is not aware of the details of how the asynchronous operation is executed in the background.

BackgroundWorker

- **Performs background work Able to report the Progress Able to cancel, if necessary**

CODE

```
1 void bw_DoWork(object sender, DoWorkEventArgs e) {  
2     for (int i = 0; i < filesToCopy.Length; ++i) {  
3         File.Copy(filesToCopy[i], dest, true);  
4         backgroundWorker.ReportProgress(  
5             (int)((100.0f * i) / filesToCopy.Length));  
6         if (backgroundWorker.CancellationPending) {  
7             e.Cancel = true;  
8             return; } }  
9     e.Result = filesToCopy.Length; }
```

A simple pattern for performing asynchronous work that is highly popular in the Windows Forms UI programming world is the “Background Worker” pattern. The BackgroundWorker class is a component that can be dropped to the Windows Forms design surface, and used to perform background work. It supports the ability to report progress and poll for cancellation, which makes it a compelling alternative to the APM when simple tasks are involved.

This slide shows an example of the method registered for the BackgroundWorker.DoWork event. This event is invoked when the background worker is run, and it contains the bulk of the background work that the worker was created to perform. In our example, the background worker copies files around and reports progress every time a file is copied. Additionally, it checks for cancellation after copying every file, and if the operation was cancelled – aborts the work and returns. Finally, it also reports a result by assigning a value to the Result property of the DoWorkEventArgs class which is passed to the event handler as a parameter.

As you see, the background worker pattern streamlines the work of reporting progress, supporting cancellation and retrieving the execution result.

5-24 | Module 05 - Threading and Asynchronous Programming



See the **BGWork** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how the background worker is used to copy files asynchronously without stalling a Windows Forms user interface (UI) and how cancellation is inherently supported.

Lab: APM in the WinForms UI



Picture Feed Lab:

Use the **PictureFeed_Starter** solution under the **Exercises\Module05_Threading** folder as a starter solution for implementing asynchronous picture retrieval capabilities in a picture feed display application.

The application synchronously retrieves picture names and picture data as pictures are selected, which makes the UI unresponsive for significant time intervals. You should modify the initialization code (when picture names are retrieved) and the code which retrieves a picture data when it is selected so that it does the work asynchronously.

You can find the solution for this lab in the **PictureFeed_Solution** solution under the **ExerciseSolutions\Module05_Threading** folder.

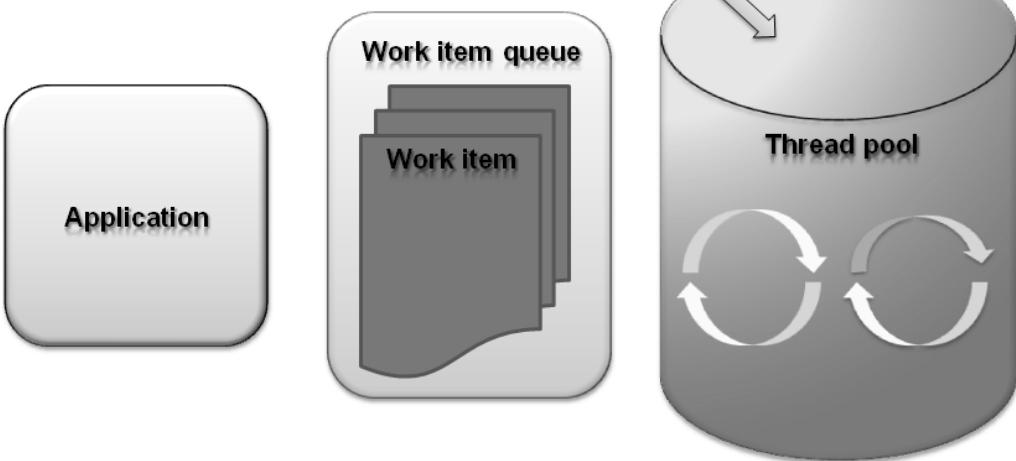
5-26 | Module 05 - Threading and Asynchronous Programming

Queuing Work for Execution



Which threads execute BeginInvoke?

Answer: Thread pool threads



In our discussion of the APM and the background worker pattern, we have neglected to mention which threads execute the actual asynchronous work. It is clear that the calling thread is absolved of the responsibility to execute the asynchronous operation – but where does “the other thread” come from?

The answer is *thread pool threads*, which are special threads managed by the .NET *thread pool* which is part of every .NET process. The thread pool is a set of threads and a queue of work items. As work items are enqueued into the work item queue, one of the thread pool threads wakes up, retrieves the work and performs it. After finishing the work, the thread is not destroyed – it returns to a waiting state and waits for more work to be enqueued. Because the thread pool must support multiple work items executing concurrently, it typically contains many threads – up to 250 threads per processor as of CLR 2.0 SP1.

ThreadPool.QueueUserWorkItem

```
1 class AsyncLogger {
2     private readonly StreamWriter _writer;
3     public AsyncLogger(string file) {
4         _writer = new StreamWriter(file);
5     }
6     public void WriteLog(string message) {
7         _writer.Write(message);
8     }
9     public void WriteLogAsync(string message) {
10        ThreadPool.QueueUserWorkItem(delegate {
11            WriteLog(message); });
12    }
13    public void Close() {
14        _writer.Close();
15    }
16 }
```

CODE

This code example shows how the .NET thread pool can be utilized explicitly (without resorting to the APM with BeginInvoke) to execute a fire-and-forget task. The ThreadPool.QueueUserWorkItem method takes a WaitCallback delegate which performs a operation. There are no built-in means to return a value from the asynchronous operation – so this is indeed a fire-and-forget asynchronous invocation.

In this example, lines 10-11 show how the WriteLogAsync method provides asynchronous semantics to writing a message to a log file. It simply queues a work item to the thread pool, and the body of the work item is an anonymous method which invokes the WriteLog synchronous operation. Because this occurs on a thread pool thread, the calling code – and the calling thread – do not stall for the completion of the WriteLog operation.

Note that although this seems like a very easy, fool-proof approach, this code is in fact riddled with concurrency bugs. For example, if multiple threads invoke the WriteLogAsync method simultaneously, then the log file might contain interleaved messages with mixed characters belonging to different threads. Alternatively, if a thread calls the Close method while another thread's asynchronous write operation is still in progress, the file is yanked under the feet of the WriteLog operation and it will cause an unhandled exception in the application. We are not yet at the point where we can properly handle synchronization, but it's never too early to appreciate the problem.

5-28 | Module 05 - Threading and Asynchronous Programming



See the **ThreadAndThreadPool** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments. In this demo, you will see how the AsyncLogger class uses the ThreadPool.QueueUserWorkItem method to enqueue work to the .NET thread pool in a fire-and-forget manner.

Manual Threading

- The thread pool manages threads



So can we?



Not recommended!

- Thread creation overhead, management subtleties

After our discussion of the `ThreadPool.QueueUserWorkItem` method and granted our understanding of the inner workings of the thread pool – a legitimate question comes to mind. Is it possible to create threads directly and manage them according to our needs? In the beginning of this module, we have seen several scenarios in which creating multiple threads is a useful feature, e.g. for better reflecting the structure of real-world interactions.

Indeed, the .NET framework provides the ability to directly create, terminate and manipulate threads. However, this is not recommended and is not the natural way of working with the managed threading and concurrency abstraction. Threads are relatively expensive to create (every thread is associated with user-mode and kernel-mode resources), quite difficult to manage and very difficult to synchronize. It's best that you use a framework which already takes care of the hassle of thread management, if possible.

Additionally, the next version of the .NET Framework (.NET 4.0) will ship with a set of extensions enabling concurrent programming which switch from a thread-based to a task-based model. In the next version of .NET, programs will be structured to work with tasks, which are abstractions of work executed by automatic thread pool threads, and not with threads directly. This abstraction provides the framework the opportunities to optimize thread-related work in ways that are not possible if you're manually creating and managing threads.

5-30 | Module 05 - Threading and Asynchronous Programming

Thread Class

- ① Thread.Start
- ① Thread states

```
1 Thread thread;
2 thread = new Thread(new ThreadStart(WriteThread));
3 thread.Start();
4
5 private void WriteThread()
6 {
7     while (!stop)
8     { ...
9     }
10 }
```

CODE

If you want to create threads manually despite our warnings, or if you're writing infrastructure-level code which serves as a thread abstraction to the rest of the application, you will need to use the `Thread` class from the `System.Threading` namespace. The single most important method is the `Thread.Start` method, which starts the execution of a thread. Prior to the call to `Start`, you initialize the thread with a delegate (parameterized or parameterless) which does the actual work of the thread. In the example on the slide, the function `WriteThread` is passed as a parameter to the `Thread` constructor on line 2, and this is the function that runs when the `Start` call completes on line 3.

When Does It End?

- ① Thread.Interrupt
- ① Thread.Abort
- ① Thread.Join

```
1 stop = true;  
2 thread.Join();
```

CODE

Threads can end in various ways – the easiest and most popular one is for the thread's method to exit (return) naturally. When this happens, the thread is terminated and is not restarted unless instructed to do so by the application.

For the calling thread to checkpoint with another thread, the Thread.Join method is often used. This method waits for another thread to complete, essentially forming a join-point in the thread execution flowchart. Other alternatives, which are not recommended for use, include the Thread.Suspend and Thread.Resume method pair, the Thread.Abort and the Thread.Interrupt methods. Later in this module we will see alternatives for inter-thread communication which are significantly safer and more convenient to use than the direct methods of the Thread class.

5-32 | Module 05 - Threading and Asynchronous Programming

Abort vs. Interrupt



When to interrupt and when to abort?

- `ThreadInterruptedException`,
`ThreadAbortedException`
- `Thread.ResetAbort`

What is the difference between `ThreadInterrupt` and `ThreadAbort`? Provided that we already know that both methods are not recommended for production use, what is their effect on the running thread?

The `ThreadInterrupt` method is only relevant if the thread is in the `WaitSleepJoin` thread state, which is achieved when the thread is blocking for something and not willing to run at the given point in time. This method breaks the thread from the `WaitSleepJoin` state and induces a `ThreadInterruptedException` within the context of the thread. This exception indicates to the thread that it was interrupted externally. However, if the thread catches this exception it can proceed normally.

The `ThreadAbort` method induces a rude asynchronous abort within the context of the target thread. Finally blocks are executed and the `ThreadAbortedException` is thrown within the target thread. If this exception is caught, it will be automatically rethrown at the end of the catch block handling it unless the `ThreadResetAbort` static method is called. Note that the `ThreadResetAbort` method requires full-trust code.

Both `ThreadInterrupt` and `ThreadAbort` induce an asynchronous exception within the context of the target thread. This makes it very difficult to write code – how can you possibly anticipate the condition where between almost any two lines of code you might receive an asynchronous exception that you didn't ask for? This is one of the primary reasons why we recommend that you never use the `ThreadInterrupt` and `ThreadAbort` methods in production quality code.

Inter-Thread Communication

- “Global” variables
- Queues

Proper inter-thread communication usually relies on a communication mechanism that is shared to both threads, i.e. stored within the virtual address space of the enclosing process. One example of such communication mechanism is a global variable – threads can use a shared global variable (if it is sufficiently protected, or if it is atomic – see below) to indicate to each other the state transitions that are occurring in the application context. Another alternative is the use of a queue – a producer thread can enqueue work items for a consumer thread to read. Here the threads are communicating using a custom mechanism, and are not relying on asynchronous exceptions or any other rude technique to tell each other what to do.

5-34 | Module 05 - Threading and Asynchronous Programming



See the **ThreadAndThreadPool** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments. In this demo, you will see how a queue can be used for communication between threads. Producers enqueue work items into a queue maintained by a logger, and the logger thread reads work items from the queue and executes them by performing the log request.

Shared Data → Synchronization

- ➊ Shared data may become corrupted

```
1 class Counter
2 {
3     private int _value;
4
5     public int Next() { return ++_value; }
6
7     public int Current { get { return _value; } }
8 }
```

CODE

As we have seen before, shared data that is used by multiple threads has the potential of becoming corrupted. This can occur for a variety of reasons. The simplest one is demonstrated in this slide (and the subsequent demo).

The slide shows a Counter class which maintains an integer value that is incremented using the Next method, and retrieved without modification by the Current property. If multiple threads use the Next method at once, it is possible for a lost update to occur and for the counter to not be updated. For example, if there are two threads, each modifying the counter ten times, it is possible for the counter to be incremented less than 20 times.

This is possible because the apparently atomic instruction “`++_value`” is in fact not atomic, and on most processors would be implemented as a sequence of three instructions. Additionally, write buffering and other hardware concerns might result in lost updates as well.

MCT USE ONLY. STUDENT USE PROHIBITED

5-36 | Module 05 - Threading and Asynchronous Programming



See the **Synchronization** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how the shared counter variable becomes corrupted – it does not receive the expected value after a lot of updates performed by multiple processors. Note that if this demo is executed on a single-processor machine, there is actually a very remote chance of a corruption occurring (a context switch has to occur exactly between the three instructions required to modify the counter). However, on a multi-processor machine the corruption is practically guaranteed.

Shared Data Races

- ➊ Non-volatile variables are subject to optimization

```
1 bool stop = false;
2 ThreadPool.QueueUserWorkItem(delegate
3 {
4     while (!stop);
5     Console.WriteLine("Done");
6 });
7 Console.ReadLine();
8 stop = true;
9 Console.ReadLine();
```

CODE

Another example of a problem that is caused by shared data is shown in this slide. In lines 3-6 you see the code for a thread which runs in a tight loop waiting for the *stop* variable to become true. The surrounding method sets the variable to true on line 8. However, if you run this example in Release mode, you might find that the thread pool thread does not actually complete its execution.

The reason for this is a compiler optimization. When the compiler sees a loop that waits for a variable to become true, it's possible to hoist the check outside the loop and convert the loop to the equivalent of a "while (true)", which never terminates. To prevent this kind of optimization, it's possible to mark the *stop* variable as volatile.

5-38 | Module 05 - Threading and Asynchronous Programming

Busy Synchronization

- **Volatile variables**
- **Interlocked.XXX**

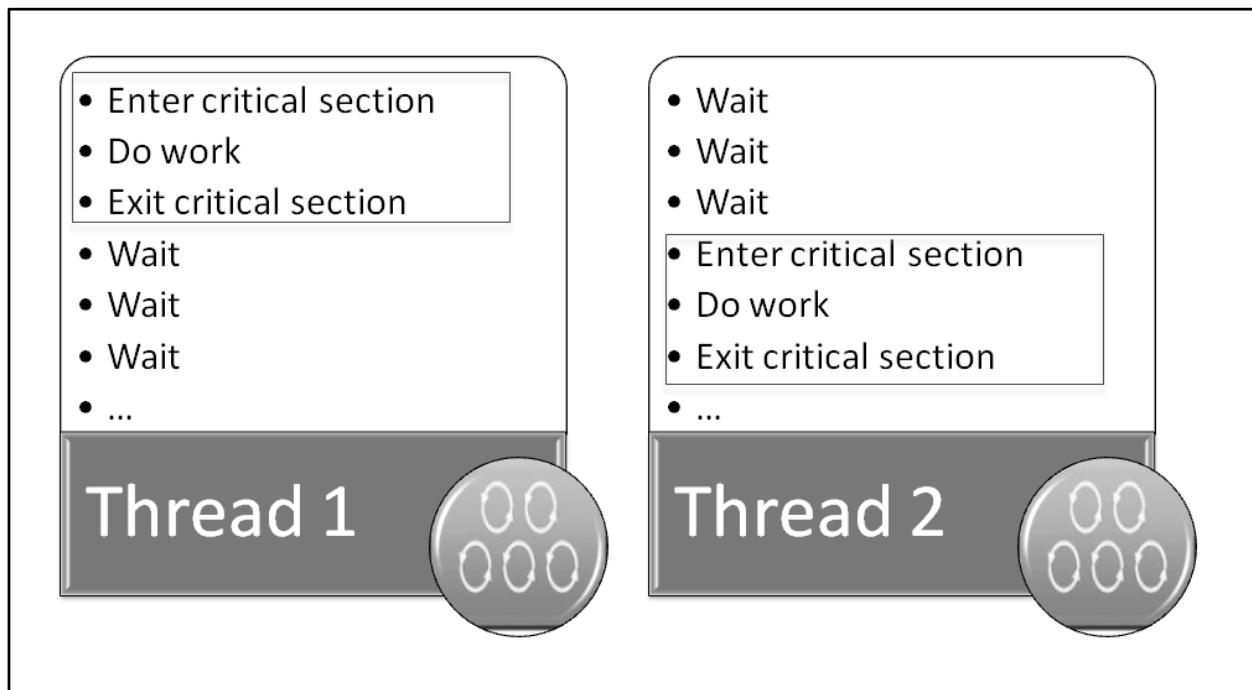
```
1 class InterlockedCounter
2 {
3     private int _value;
4
5     public int Next()
6     {
7         return Interlocked.Increment(ref _value);
8     }
9
10    public int Current { get { return _value; } }
11 }
```

A large, stylized, three-dimensional white text logo with a black shadow effect, spelling the word "CODE".

The Interlocked family of operations (implemented in the Interlocked static class) provide the facilities for atomic operations on a fairly limited set of types. One of these operations is an atomic increment of a variable – while the atomic increment occurs, it is guaranteed that no other thread will see the old value, resulting in a lost update.

Therefore, a synchronized and working version of the Counter class shown a few slides ago is here on the slide, and it's using the Interlocked.Increment method to guarantee synchronization. Working with interlocked operations is outside the scope of this module and this course, and it is in fact extremely difficult. (For more information, we recommend the Windows Concurrent Programming course.)

Critical Sections



An alternative to “busy” synchronization, which is induced by the interlocked operations or by waiting for a global variable, is to use a critical section. A critical section is a section of code in which only one thread can be at a given time. On the slide, two threads are trying to enter the critical section. The first one succeeds and does some work, while the other thread waits. When the first thread releases (exits) the critical section, the second thread enters it, again preventing everyone else from entering the critical section.

If all shared data is protected by critical sections, we can guarantee correctness because the data transformations can be proven equivalent to a serialized (single-threaded) execution. However, there are severe scalability problems with using critical sections to protect the entire set of shared data, due to Amdahl’s law – the more code does not yield to parallelization, the smaller the returns on adding processors to the system.

This tradeoff between correctness and scalability is not the first ever observed in computer science.

5-40 | Module 05 - Threading and Asynchronous Programming

Monitor and Lock

- **Monitor.Enter and Monitor.Exit**
- **The lock keyword also performs the same function.**

```
1 class BankAccount {  
2     public decimal Balance { get; private set; }  
3     private readonly object _syncRoot = new object();  
4     public void Deposit(decimal amount) {  
5         lock (_syncRoot)  
6             Balance += amount;  
7     }  
8     public void Withdraw(decimal amount) {  
9         lock (_syncRoot)  
10            Balance -= amount;  
11    } }
```

A large, stylized, three-dimensional white text "CODE" with a black shadow effect.

The Monitor class (which resides in the System.Threading namespace) can be used to implement a critical section in .NET. A language alternative, the *lock* statement, can be used in C# for an equivalent of the same. (*lock* statements are translated to Monitor.Enter and Monitor.Exit method call pairs.)

In the example on the slide, a bank account's balance is protected from concurrent modification by using a *lock* statement on the *_syncRoot* private object. The code to modify an account's balance in the Deposit and the Withdraw method is protected by the lock statement, and therefore only one thread will be able to modify the balance at any given time.

Other alternatives to easy synchronization include the

[MethodImpl(MethodImplOptions.Synchronized)] which can be placed on any static or instance method of a type. If all the class methods are using this attribute, only a single thread will be able to enter the class' methods at a given time. This provides an easy synchronization solution but often is too coarse-grained and provides a significant scalability hit.

Pay attention to the use of a private synchronization object instead of using the *this* reference (or the *typeof(BankAccount)* reference, in a static method) for locking. Although it still appears in some outdated MSDN samples, it is not recommended to use public objects for synchronization – exactly because they are public and other code might use them for synchronization (perhaps inadvertently).

Synchronization with
lock

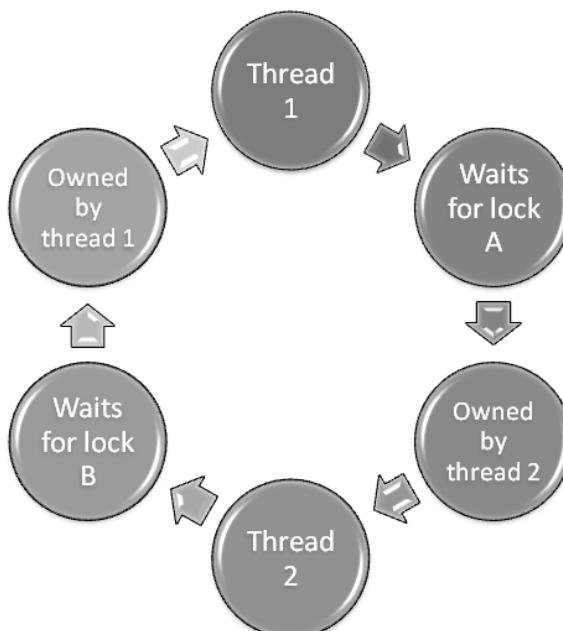


See the **Synchronization** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how synchronization is used to protect shared data from concurrent modification.

5-42 Module 05 - Threading and Asynchronous Programming

Deadlocks



Deadlocks are one of the nastiest problems of synchronization. A deadlock (also known as “deadly embrace”) occurs when threads in the system wait for each other, making no forward progress. (There is also the term *livelock*, meaning that the threads in the system are not waiting – they are doing work – but they are not making any progress.)

The simplest deadlock is a thread waiting for itself. A more complicated deadlock is depicted in the diagram on the slide. Start reading the diagram from Thread 1 – it waits for a lock that is owned by thread 2, but thread 2 waits for a lock owned by thread 1. Until thread 1 releases the lock, thread 2 can't make any progress. Until thread 2 makes progress, thread 1 can't release the lock. This is a mutual deadlock and the system is stuck – the only way out is to terminate one of the threads or to abandon one of the locks – an action that will have adverse effects on the rest of the system.

There are no silver bullets – there is no absolute solution for the problem of deadlocks. One idea revolves around timeouts – if all wait operations are issued with a finite timeout, and a graceful abort is attempted after the timeout period, then deadlocks will not occur. Another idea has to do with lock leveling – if all locks (or resources) in the system are acquired in the same order (e.g. first lock A, then lock B and so on), deadlocks will not occur. Finally, there are also more theoretical ideas such as the Banker's Algorithm, which we will not discuss in this module.

Deadlocks in Code

```
1 Counter c1 = new Counter();
2 Counter c2 = new Counter();
3 ThreadPool.QueueUserWorkItem(delegate {
4     lock (c1) {
5         for (int i = 0; i < 100000; ++i) c1.Next();
6         lock (c2) {
7             for (int i = 0; i < 100000; ++i) c2.Next();
8         }
9     });
10 ThreadPool.QueueUserWorkItem(delegate {
11     lock (c2) {
12         for (int i = 0; i < 100000; ++i) c2.Next();
13         lock (c1) {
14             for (int i = 0; i < 100000; ++i) c1.Next();
15         }
16     });
}
```

CODE

The code on this slide demonstrates a simple deadlock occurring in code. There are two threads (thread pool threads) which performing locking in a different order – the first thread locks *c1* first and then locks *c2*, and the second thread does so in the opposite order. This almost definitely results in a deadlock.

5-44 | Module 05 - Threading and Asynchronous Programming



See the **Synchronization** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how deadlocks occur and discuss some practical ideas how they can be prevented.

Pulse, PulseAll, Wait, WaitAll

```
1 class WorkQueue<T> : Queue<T> {
2     private readonly object _sync = new object();
3     public new void Enqueue(T item) {
4         Monitor.Enter(_sync);
5         try {
6             base.Enqueue(item);
7             Monitor.Pulse(_sync); } finally {
8                 Monitor.Exit(_sync);
9             } }
10    public new T Dequeue() {
11        Monitor.Enter(_sync);
12        try {
13            while (base.Count == 0) Monitor.Wait(_sync);
14            return base.Dequeue(); } finally {
15                Monitor.Exit(_sync);
16            } } }
```

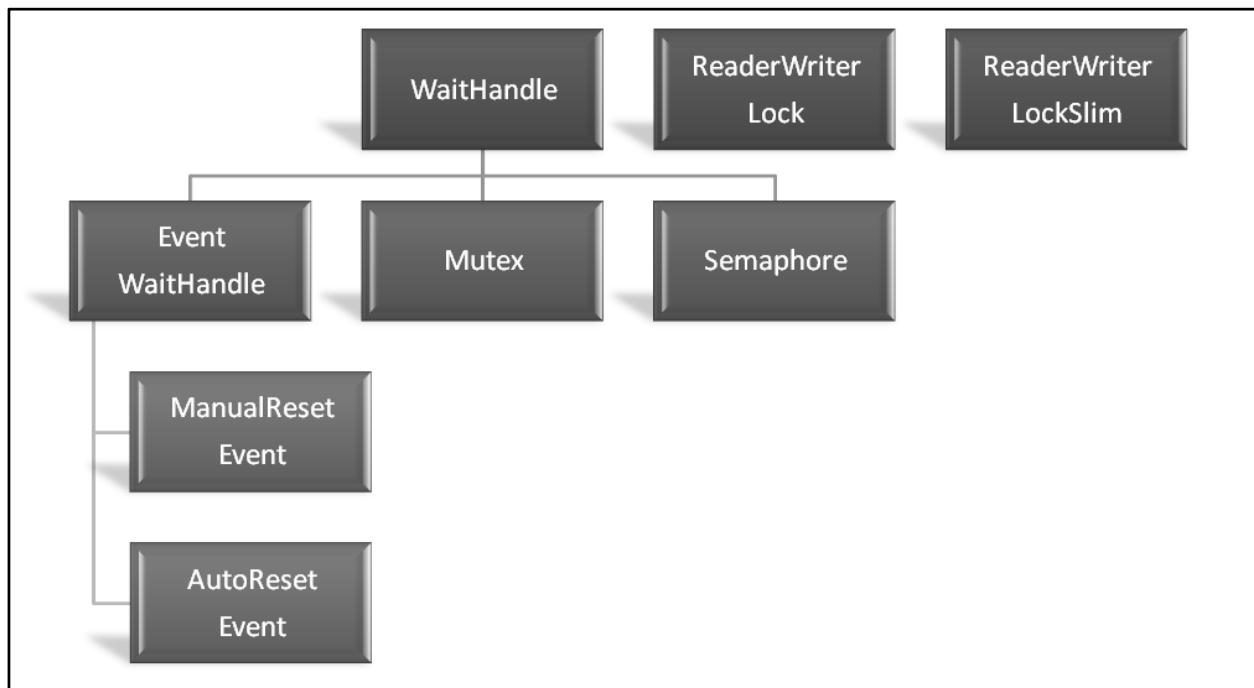
CODE

The Monitor class features advanced methods for implementing sophisticated concurrency design patterns (for more information on concurrency design patterns, consider reading <http://blogs.microsoft.co.il/blogs/sasha>). The Pulse, PulseAll, Wait and WaitAll methods of the Monitor class are an implementation of a condition variable, and are used in the slide to implement a producer-consumer queue.

The code shows a WorkQueue generic class which derives from the generic queue and provides new versions of the Enqueue and Dequeue operations. The enqueue operation takes a lock, but after enqueueing a work item it will use the Monitor.Pulse method to signal to consumers that work is available. The dequeue operation will use the Monitor.Wait method to wait for work become available if the queue is empty. After the Wait method returns, the dequeue operation is guaranteed to own the lock and can proceed to dequeue an item from the queue safely. Note that in this example, the producer and consumer do not have to use any sort of special communication mechanism, and do not have to resort to busy waiting (e.g. if the consumer detects that the queue is empty). This makes Monitor a very powerful advanced synchronization mechanism.

5-46 Module 05 - Threading and Asynchronous Programming

WaitHandle Synchronization



A significant alternative to the Monitor class and the related *lock* statement is the hierarchy of synchronization objects which reflect quite closely the state of Windows synchronization mechanisms (exposed by the operating system). At the root of the hierarchy we find the WaitHandle abstract class, which provides the fundamental ability to wait for a synchronization object to become signaled.

Derived classes, including the Mutex, the Semaphore and various kinds of event implementations, override the meaning of waiting for an object to become signaled and override the semantics of the synchronization mechanism. For example, while a mutex is an advanced sort of critical section (which works across processes as well as across threads), a manual reset event is primarily a notification mechanism which allows one thread to wait for an event to occur in the system.

The intrinsics and semantics of Windows synchronization mechanisms are outside the scope of this course, but they are well covered in the MSDN documentation of the underlying Win32 mechanisms and of the .NET wrapper classes depicted in this diagram. (Alternatively, refer to the Windows Concurrent Programming course for more information.)

Using Events for Synchronization

```
1 public static void DoAndLetMeKnow(  
2             Action action, EventWaitHandle @event)  
3 {  
4     ThreadPool.QueueUserWorkItem(delegate  
5     {  
6         action();  
7         @event.Set();  
8     });  
9 }
```

CODE

In this slide, an EventWaitHandle-derived object is used to implement an extension of the thread pool's fire-and-forget semantics. Unlike the standard QueueUserWorkItem method provided by the ThreadPool class, this method (called DoAndLetMeKnow) will perform an action using the thread pool but signal a synchronization mechanism (an event) when the action completes. This allows the caller to wait for the event if appropriate, and models in good similarity the AsyncWaitHandle property of the IAsyncResult interface, which we have seen when discussing the asynchronous programming model (APM) built into the .NET delegates.

5-48 | Module 05 - Threading and Asynchronous Programming



See the **Synchronization** project in the **Module05_Threading** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how advanced synchronization mechanisms (outside the scope of a lock and Monitor) can be used to implement synchronization, and how concurrency design patterns rely on advanced synchronization mechanisms.

Parallel Extensions for .NET

- `Parallel.For`, `Parallel.ForEach`
- Parallel LINQ
- Tasks, futures

The Parallel Extensions for .NET, which will be released as part of the next .NET version - .NET 4.0 – will contain advanced support for synchronization and concurrency work using the .NET framework. Focusing on implicit parallelism, the Parallel Extensions will also contain fundamental low-level classes for using concurrent thread-safe collections, for interacting with tasks and not threads, for implementing advanced concurrency design patterns and for many other uses.

Implicit parallelism is best demonstrated by the idea of a parallel for-each operation – instead of performing the for-each work sequentially (serially), an automatic scheduler can distribute some iterations of the loop to multiple threads, improving the throughput of the operation.

This kind of parallelism is called implicit because there is no additional work taken on the programmer's side other than converting the foreach loop to a `Parallel.ForEach` method call.

Another example is Parallel LINQ (PLINQ) – by adding the `.AsParallel()` clause to the end of the data source in a LINQ query, the scheduler automatically executes the query in parallel.

Other mechanisms are more explicit – for example, tasks represent a work item that is executed at a later time by the scheduler; a future represents a value that is not currently known and will be calculated in the background; and there are many others.

To learn more about the Parallel Extensions for .NET, visit the Microsoft Parallel Computing Center on MSDN.

5-50 | Module 05 - Threading and Asynchronous Programming

Lab: Thread-Safe Resource Parallelizing Work



Queuing Work Lab:

Use the **QueuingWork_Starter** solution under the **Exercises\Module05_Threading** folder as a starter solution for implementing a thread-safe queue of items, and using it to implement a single-producer/multiple-consumer scenario for calculating the word counts of lines in a file. The program creates a varying number of consumer (processor) threads which perform the calculations using a pair of input and output thread-safe queues.

You can find the solution for this lab in the **QueuingWork_Solution** solution under the **ExerciseSolutions\Module05_Threading** folder.

Summary

- Multi-tasking, processes, threads, asynchrony, scheduling
- Asynchronous programming model (APM)
- Thread pool
- Manual threading
- Synchronization
- Overview of Parallel Extensions for .NET
- Lab



5-52

Module 05 - Threading and Asynchronous Programming

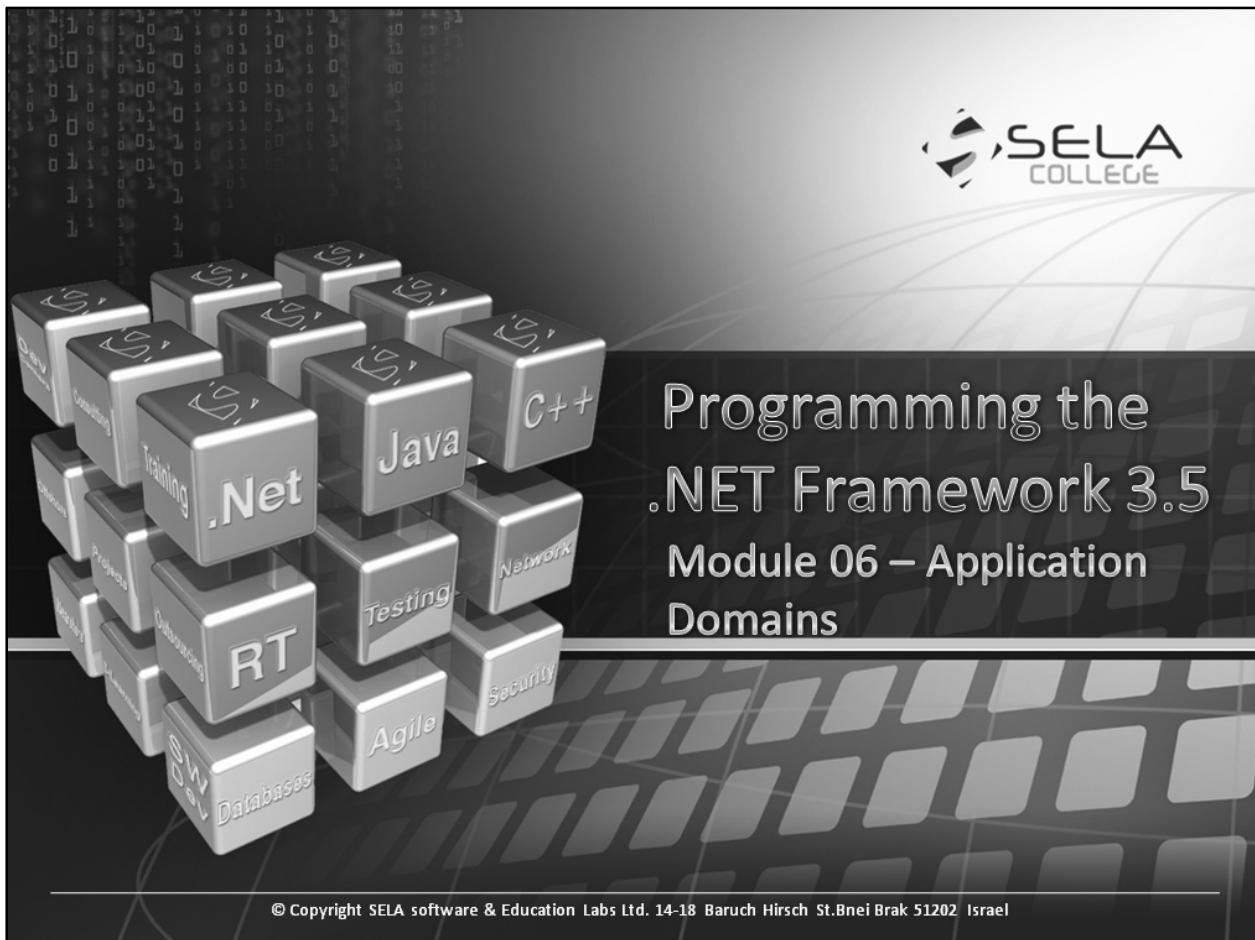


Module 06 - Application Domains

Contents:

In This Chapter.....	3
System Isolation Boundaries.....	4
Application Domain Isolation.....	5
Why AppDomains?	6
Properties of an AppDomain.....	8
Things to Look Out for.....	9
Creating an AppDomain.....	10
Retrieving Assemblies.....	11
Unloading an AppDomain	12
Executing Code in an AppDomain	13
Crossing AppDomain Boundaries	15
Marshal-by-Value	16
Marshal-by-Reference.....	17
.NET Remoting Overview.....	19
Lab: Plugin Framework	21
Summary	22

6-2 | Module 06 - Application Domains



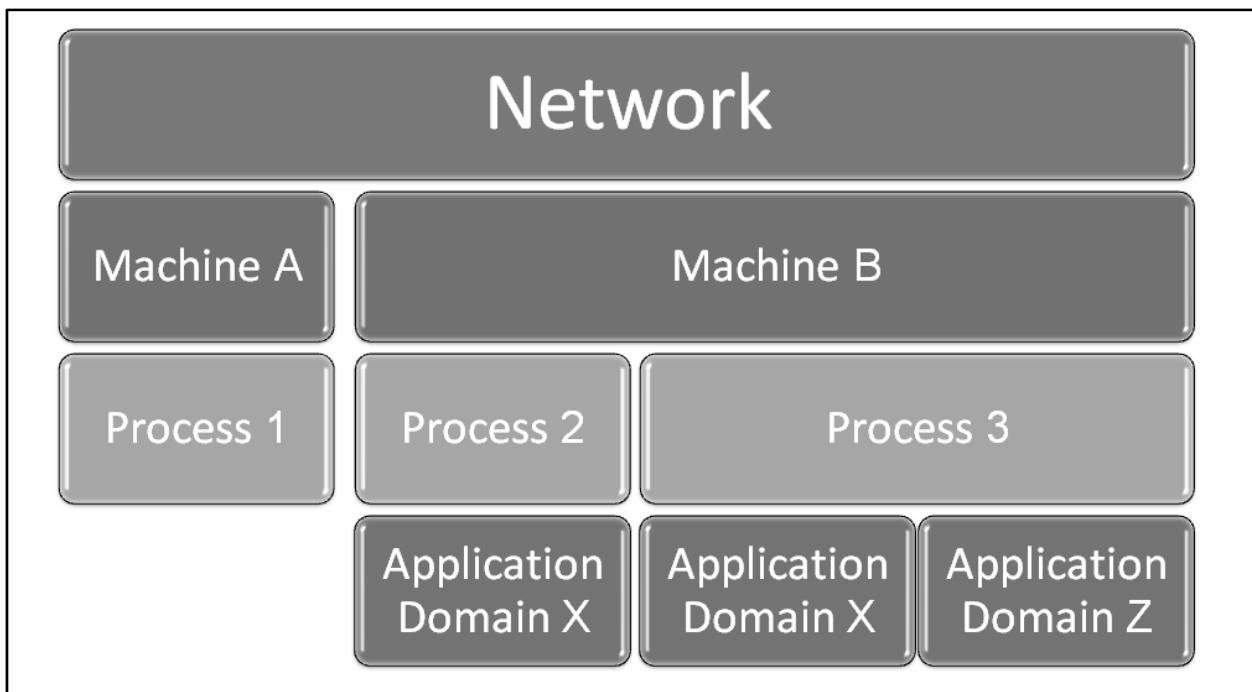
In This Chapter

- ⌚ AppDomains as isolation boundaries
- ⌚ Creating and unloading AppDomains
- ⌚ Executing code in an AppDomain
- ⌚ AppDomain boundaries
- ⌚ Overview of .NET Remoting
- ⌚ Lab



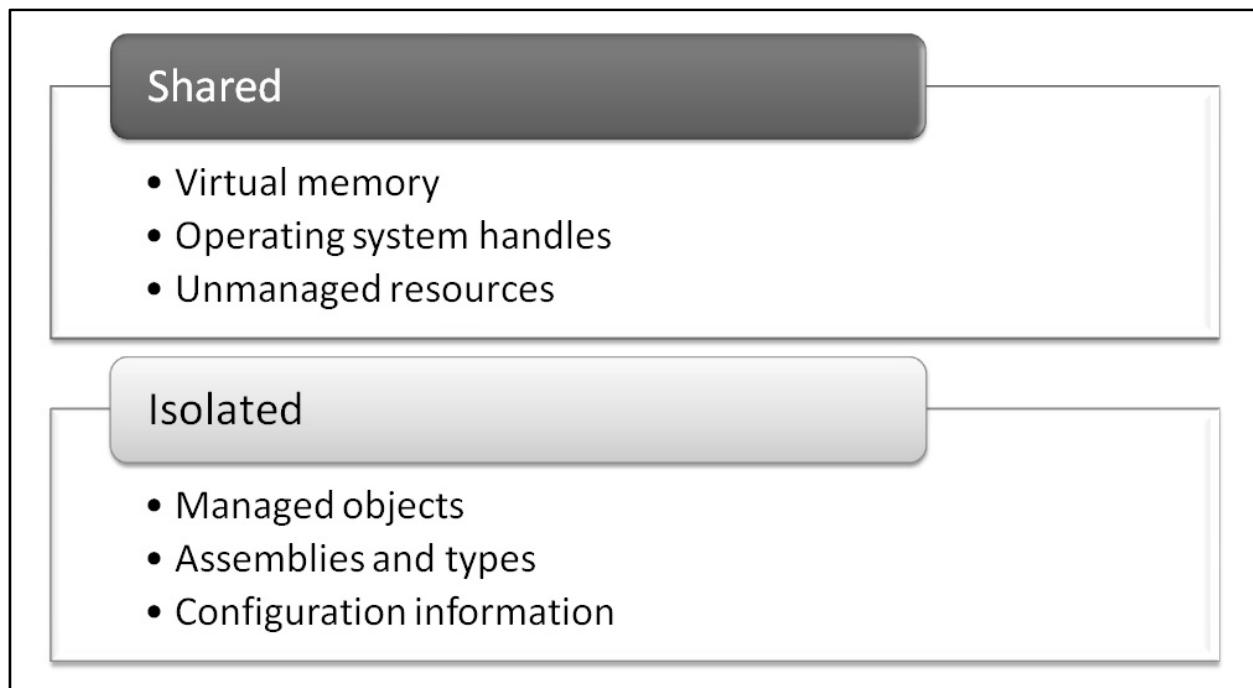
6-4 | Module 06 - Application Domains

System Isolation Boundaries



Applications execute within a multitude of isolation boundaries. Some of these boundaries are enforced by the operating system (kernel-mode and user-mode separation, separate processes), some are enforced by machine boundaries, yet others are enforced by network administrators on network boundaries. Isolation boundaries provide various useful features including reliability, fault isolation, security, versioning and dependency management (we will look into these features later in this module). The isolation boundary for managed applications can be more granular than the operating system process – there can exist multiple isolation boundaries within a single OS process, called application domains. In this module, we will examine the role of application domains and how communication ensues between application domains.

Application Domain Isolation



What is the degree of isolation that application domains provide to a managed application? Managed code always executes within the boundaries of the operating system process. This implies that resources shared on the OS process level are shared between application domains, including virtual memory, operating system handles and other unmanaged resources that do not get the attention of the .NET runtime. However, because a managed application *usually* interacts with virtual memory, OS handles and unmanaged resources through the CLR, a degree of isolation is possible even though these resources are shared within the process. For example, even though corrupting the process' memory space is something that cannot be isolated from other application domains, the type safety of the .NET runtime asserts that memory corruptions are impossible without cooperation with unsafe or unmanaged code.

The resources that are isolated and shared only within the application domain's boundaries include managed objects (even static members), assemblies and the types contained in these assemblies, and the application domain's configuration information (more on it later in this module). Although managed objects are not shared between application domains by default, it is possible to pass managed objects across application domain boundaries (we will see how to do so later). The ability to configure an application domain differently from the rest of the application (with regard to a variety of configuration aspects) provides essential feature to side-by-side code execution, which we will examine next.

6-6 | Module 06 - Application Domains

Why AppDomains?

- Reliability (fault isolation)
- Dynamic code unloading
- Security
- Configuration
- Versioning

An application domain provides a light-weight isolation boundary that is by an order of magnitude more scalable than the isolation provided by operating system processes. By using application domains to isolate parts of an application or individual applications within a single process, a managed application gains the following advantages:

- Reliability (fault isolation) – type-safe code running in a separate application domain will not corrupt the state of the entire process. For example, when a web browser must load third-party application components (add-ins) and execute them, there is no way of knowing whether these add-ins are as reliable as their host process. Running each add-in in a separate application domain ensures that add-ins cannot corrupt each other's state, and that add-ins cannot corrupt the state of the entire host. A fault in an application domain can be managed so that only the application domain will be terminated, without affecting the state of the rest of the application.
- Dynamic code unloading – application domains provide a code-unload boundary that is otherwise impossible to obtain. The CLR does not provide a facility for unloading an individual assembly (akin to the Win32 FreeLibrary API) – only an entire application domain can be unloaded. For long-running applications, this is a critical scalability and reliability feature because code can be loaded to execute for a short period of time and then unloaded with its application domain. This is also highly useful for versioning scenarios (see below).
- Security – code executing in a separate application domain can be assigned a limited set of permissions, allowing for secure execution of untrusted code in a trusted application.
- Configuration – an application domain can have its own set of configuration information, including a separate default application configuration file, a different base directory for assembly loading, and other configuration traits.
- Versioning – application domains allow multiple versions of the same assembly to execute concurrently within the same process by isolating each version into a separate application

Module 06 - Application Domains | 6-7

domain. Long-running applications or applications that require side-by-side versioning will greatly benefit from this ability.

Properties of an AppDomain

- Friendly name, id
- Base directory
- Application configuration
- Security policy
- Shadow copy configuration

When an application domain is created, it is associated with a set of setup information (from the *AppDomainSetup* class). This setup information affects the characteristics of the application domain which are not necessarily shared with its creating domain or the application's default domain. Among these characteristics:

- An identifier (assigned by the runtime) and a friendly name (which the client can control)
- A base directory for loading assemblies and executing code
- An application configuration file
- Security policy including permissions assigned to code that executes within the application domain
- Shadow copy configuration, enabling run-time replacement of assemblies used by code in the application domain by shadowing (copying) the assemblies to a private directory before loading them

Most of the time, you would not use all setup properties on an application domain.

Nonetheless, these properties are a large part of the isolation boundary abstraction that an application domain provides.

Things to Look Out for

- Static data is domain-private
- Not every object can cross domain boundaries

There are two common mistakes made when programming application domains explicitly. Both have to do with the degree of isolation that application domains provide. First, while it is intuitive that instance data is application domain private, even static data is not shared between application domains. In fact, it is possible that a specific assembly (and its types) are not even loaded within an application domain, let alone shared with other domains.

Second, not every instance can cross application domain boundaries. In this module we will examine two types of objects – marshal-by-value (serializable) and marshal-by-reference objects – which can cross the application domain boundary in different ways. However, objects that did not opt-in to be transmitted across application domain boundaries do not have this ability inherently enabled. This is a crucial point to bear in mind when troubleshooting and diagnosing communication problems between application domains. For example, attempting to pass a delegate that points to an anonymous method across domain boundaries might sometimes succeed and sometimes fail, depending on whether the anonymous method requires access to an instance during its compilation process. If the anonymous method is compiled to a simple static method, then (granted that the assembly is present on the other side of the domain boundary) it can be passed around without any problems. However, if the anonymous method is compiled to a class, then the class must be marked as serializable or marshal-by-reference to be passed across the boundary. Because the programmer does not have control over the anonymous method's backing class (generated by the compiler), passing such a method across application domain boundaries is impossible.

6-10 | Module 06 - Application Domains

Creating an AppDomain

⌚ AppDomain.CreateDomain

```
1 AppDomain domain =
2     AppDomain.CreateDomain("MyFirstDomain");
3 Console.WriteLine("Base directory: " +
4     domain.BaseDirectory);
5 Console.WriteLine("Id: " + domain.Id);
6 Console.WriteLine("Configuration file: " +
7     domain.SetupInformation.ConfigurationFile);
```

CODE

Creating an application domain involves a call to the static *AppDomain.CreateDomain* method. This method has several overloads accepting a variety of configuration parameters, but the bare minimum is the application domain's friendly name (used primarily for display purposes). After creating the domain in line 1, this code example proceeds to query and output some useful properties of the *AppDomain* class, such as the base directory where the domain looks for assemblies and files, the domain numeric identifier and the domain default application configuration file.

Retrieving Assemblies

⌚ AppDomain.GetAssemblies

```
1 Array.ForEach(  
2     domain.GetAssemblies(), Console.WriteLine);  
3 //Output:  
4 //mscorlib, Version=2.0.0.0, Culture=neutral,  
5 //PublicKeyToken=b77a5c561934e089
```

CODE

Each application domain may have access to a different set of assemblies, and not every assembly used by the application will be loaded into every application domain. The *AppDomain.GetAssemblies* instance method retrieves an array of *Assembly* objects representing the assemblies loaded into the application domain. Oftentimes a communication problem between code on different sides of a domain boundary has to do with the fact that one of the sides does not have access to an assembly which contains the type being passed across the boundary. Because application domains can be configured with different security policies and different base directories for assembly loading, this scenario may occur quite frequently. Note that the main BCL assembly, *mscorlib*, is always loaded into every application domain (the assembly itself is marked as application domain *neutral*). Application domain neutral assemblies are outside the scope of this module.

6-12 | Module 06 - Application Domains

Unloading an AppDomain

① AppDomain.Unload

```
1 AppDomain.Unload(domain);
2 domain.GetAssemblies();
3 //Output:
4 //Unhandled Exception:
5 //System.AppDomainUnloadedException:
6 //Attempted to access an unloaded AppDomain.
7 // at System.AppDomain.GetAssemblies()
```

CODE

After using an application domain, it can be unloaded by calling the static *AppDomain.Unload* method. Unloading an application domain releases all resources associated with it, including all the assemblies that were loaded by that domain. Moreover, all threads executing code within the application domain are aborted so that the application domain can be timely unloaded. Using the application domain object or attempting to execute code within the application domain will result in an *AppDomainUnloadedException* exception, as demonstrated in this slide. Note that unloading an application domain is the only way to unload an assembly (there is no way to unload an individual assembly after it has been loaded). This is an extremely useful feature for long-running applications.

Executing Code in an AppDomain

- AppDomainInitializer
- ExecuteAssembly
- CreateInstanceAndUnwrap
- DoCallback

There are three primary ways to execute code within an application domain after it has been created. The following demo will show how these methods can be used in practice.

- The first way involves using an AppDomainInitializer delegate which is passed to the AppDomain.CreateDomain method when the application domain is created. The initializer delegate is invoked when the application domain is initializing, and it executes within the target application domain. While it is possible to pass an array of strings as parameters to the initializer delegate, there is no way to obtain a return value using this approach.
- The second way involves using the AppDomain.ExecuteAssembly instance method, which takes the details of an assembly, locates the Main method (entry point) within that assembly and executes it. It is fairly difficult to obtain return values or pass sophisticated parameters using this method.
- The third way involves creating an instance of the specified type within the application domain and retrieving a reference to it. If the type is marshal-by-value, this creates a local copy of the object in the calling domain. If the type is marshal-by-reference, a proxy to the target object is returned. (We will examine this in greater detail later in this module.)
- The fourth way involves using the AppDomain.DoCallback instance method which takes a delegate (with no parameters and a void return value) and executes it within the context of the application domain. The type of the delegate's target and method will be loaded in the target application domain.

MCT USE ONLY. STUDENT USE PROHIBITED

6-14 | Module 06 - Application Domains

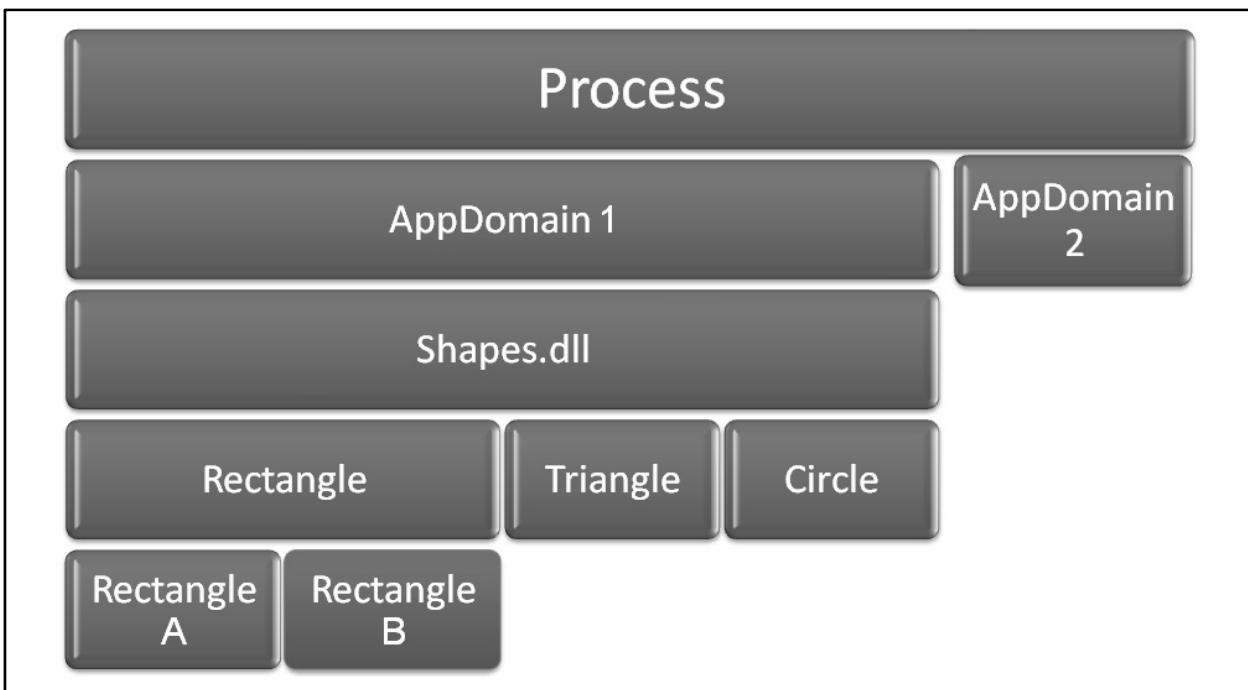
Executing Code in an AppDomain



See the **CreatingAppDomains** project in the **Module06_AppDomains** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how application domains are created, how their properties are used and how they are unloaded. You will also see the three different ways to execute code in a separate application domain that were discussed previously in this module (using an application domain initializer delegate, using the ExecuteAssembly method, using the CreateInstanceAndUnwrap method and using the DoCallback method).

Crossing AppDomain Boundaries



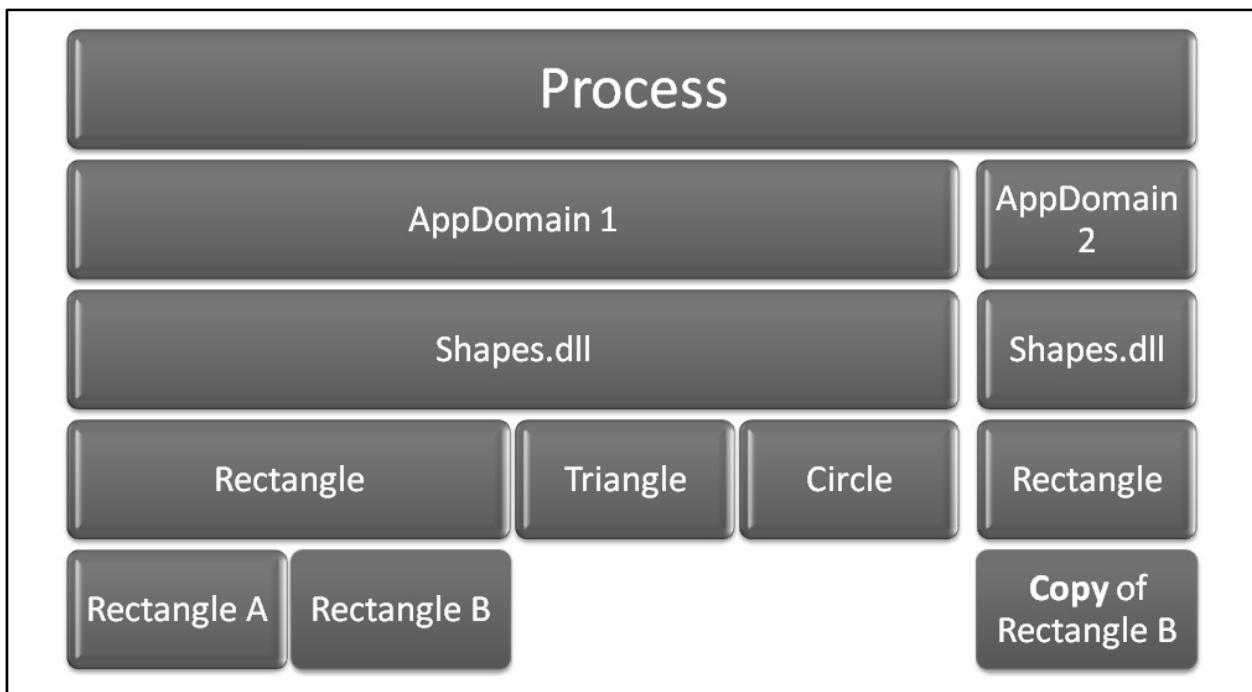
This diagram demonstrates the dilemma of passing an object between application domains. In this diagram, we have two application domains in a single process. One of the application domains has a Shapes.dll assembly loaded, containing a few types. Two instances of the Rectangle type also exist within the application domain.

How can we pass Rectangle B to be used in the second application domain?

Well, first of all, using a type requires that the type be loaded in the other application domain. So the type's assembly must be loaded in the other application domain, which means it must be discoverable and loadable. If it can be loaded, the question of MBV vs. MBR arises.

6-16 | Module 06 - Application Domains

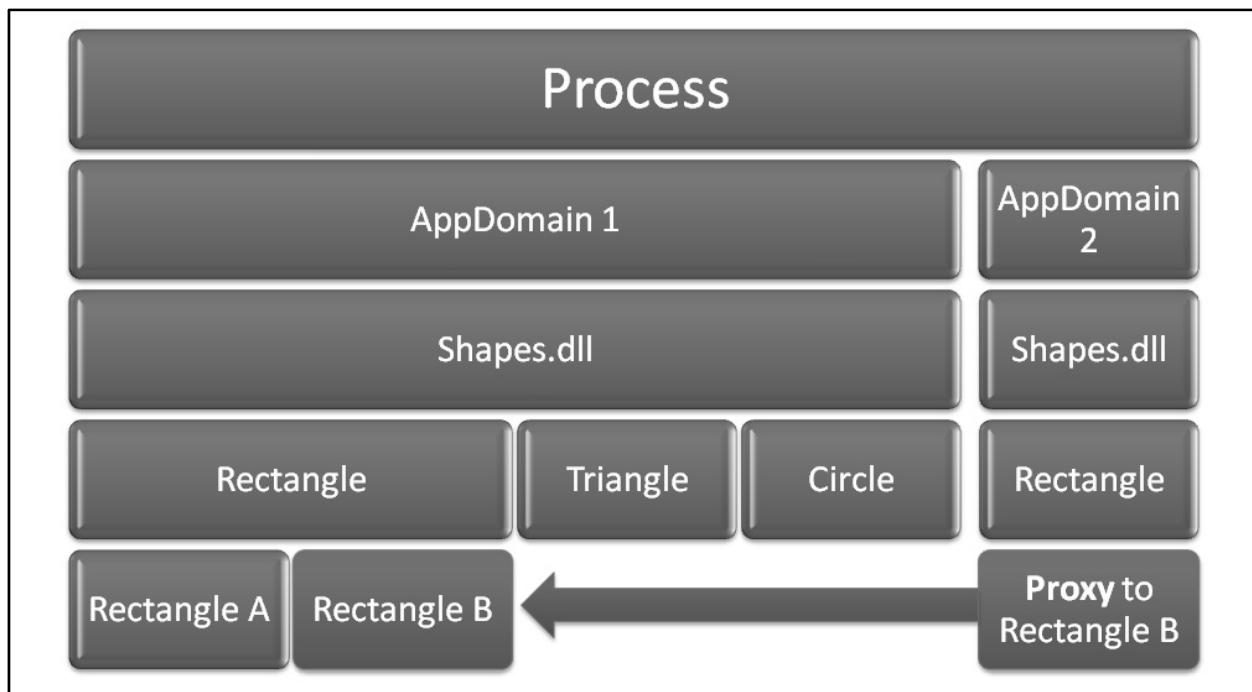
Marshal-by-Value



Marshal-by-value means that when the instance is passed across application domain boundaries, it is serialized on the caller's side and deserialized on the callee's side. To accomplish this, the instance's type must be marked as [Serializable]. The application domain infrastructure automatically takes care of serializing and deserializing the object as necessary to pass it through to the other side.

The two instances of Rectangle B in this diagram exist independently; changes made to the object in the first application domain will not affect the copy in the second application domain, and vice versa. (The following demo will demonstrate this in practice.)

Marshal-by-Reference



Marshal-by-reference means that the value of the instance is not passed across application domains; instead, a reference to it is passed without performing serialization. The object's type must still be loaded in the other application domain (along with the type's assembly, of course), but the contents of the instance are not passed across the boundary. This means that changes made to the instance will affect the reference to it in the other application domain and vice versa, as the following demo will demonstrate.

Marshaling objects by reference is an extremely interesting characteristic of inter-domain communication which is useful in wider (out-of-process) scenarios as well. For example, a distributed system can easily be implemented by marshaling out-of-process and even out-of-machine references to objects. This is called .NET Remoting and is outside the scope of this module (although a brief overview of .NET Remoting appears at the end of this module).

Making an object marshal-by-reference involves inheriting its class from the *MarshalByRef* abstract class, which is the base of all marshal-by-reference objects. It contains several methods which are outside the scope of this module and affect the behavior of the object and its proxies (external references).

6-18 | Module 06 - Application Domains

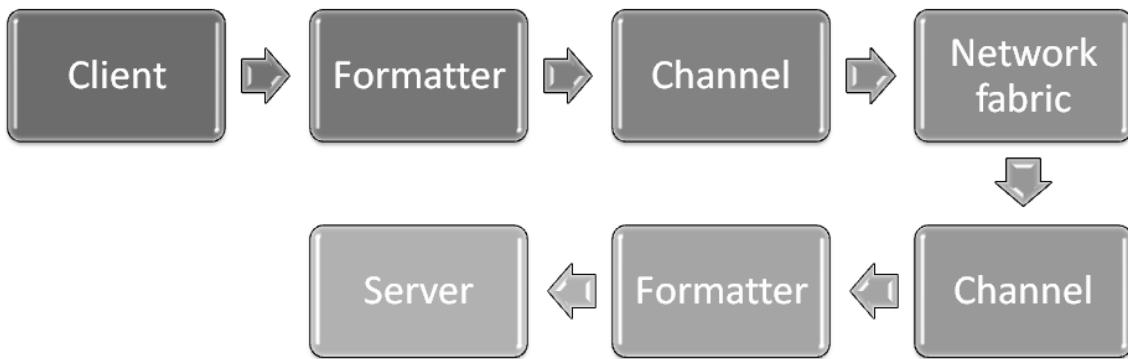


See the **MarshalingDifferences** project in the **Module06_AppDomains** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see the difference in behavior between a marshal-by-value and a marshal-by-reference instance. The marshal-by-value instance is copied across the application domain boundary, while the marshal-by-reference instance is passed by reference (using a proxy) to its caller.

.NET Remoting Overview

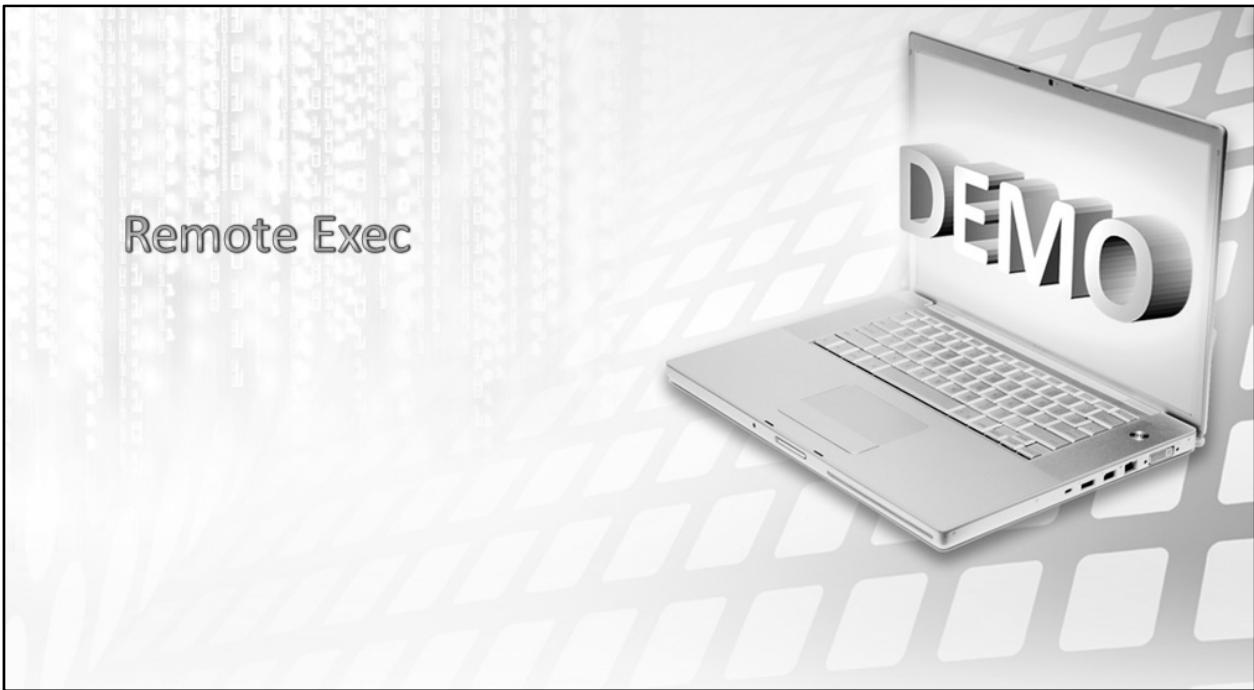
- Distributed computing infrastructure
- The same framework used by MBR



.NET Remoting is the logical extension of the communication mechanisms that exist between application domains to the context of inter-process and inter-machine communication. A distributed system implemented using .NET Remoting relies on the existence of channels and formatters. Channels implement the communication details across the network fabric, while formatters deal with the serialization of messages that are passed across the communication boundary. Marshal-by-value and marshal-by-reference are the two ways to pass data across a distributed system using .NET Remoting. The supported communication fabrics include HTTP, TCP and named pipes (local machine IPC), and the supported formatting mechanisms include SOAP and binary serialization.

A comprehensive overview of .NET Remoting is outside the scope of this module. However, it's important to understand that .NET Remoting is the infrastructure that is also used for communication between application domains. If you have a thorough understanding of marshal-by-value and marshal-by-reference, you are well-prepared to embrace the network and inter-process aspects of .NET Remoting. Because this is a technology that is no longer recommended for distributed application development (superseded by Windows Communication Foundation in .NET 3.0), this module does not address it in more detail.

6-20 | Module 06 - Application Domains



See the **RemoteExec.Shared**, **RemoteExec.Server** and **RemoteExec.Client** projects in the **Module06_AppDomains** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how a remote execution client-server system is implemented using .NET Remoting. By designing the interface between the client and the server (which allows remote execution of commands), our use of .NET Remoting abstracts away the gory communication and message passing details. A method call across the communication boundary is completely transparent, and very similar to calls across application domains. The demo also demonstrates details which are outside the scope of this module, such as the establishment of a remoting channel and the creation of server-activated singleton objects.

Lab: Plugin Framework



Plugin Framework Lab:

Use the **PluginFramework_Starter** solution under the **Exercises\Module06_AppDomains** folder as a starter solution for implementing a plugin application which executes aggregation operations on a set of double-precision numbers.

The application hosts each aggregation plugin assembly (arithmetic operations, such as sum and product; statistics operations, such as median and mean; etc.) in a separate application domain to maximize the reliability of the host. It then invokes each plugin to perform the aggregation on a set of random numbers.

You can find the solution for this lab in the **PluginFramework_Solution** solution under the **ExerciseSolutions\Module06_AppDomains** folder.

Summary

- AppDomains as isolation boundaries
- Creating and unloading AppDomains
- Executing code in an AppDomain
- AppDomain boundaries
- Overview of .NET Remoting
- Lab





Module 07 - Interoperability

Contents:

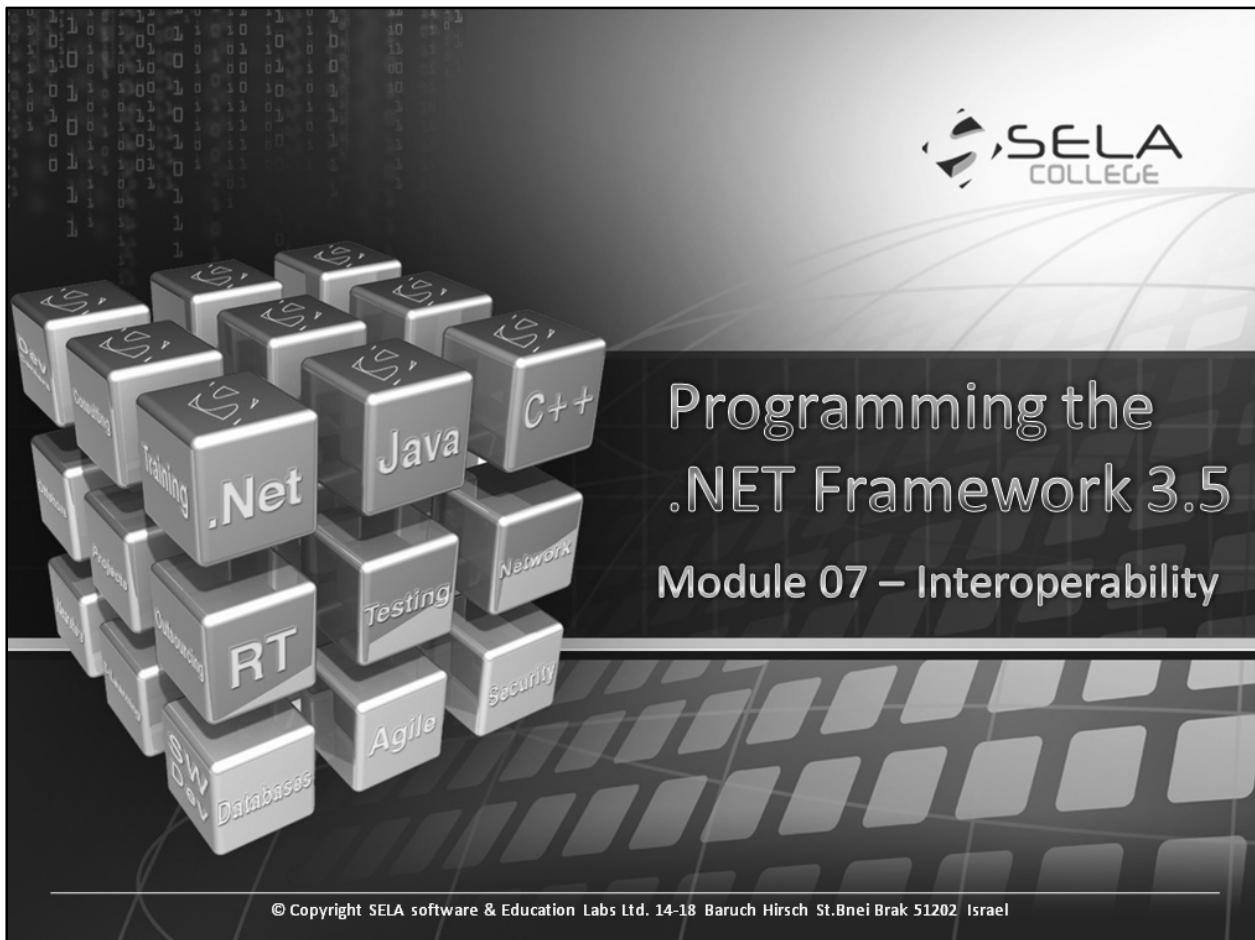
In This Chapter.....	5
Overview of Interoperability.....	6
Platform Invoke	7
COM Interoperability	8
C++/CLI	9
P/Invoke	10
Behind the Scenes	11
Marshaling.....	13
Standard Mappings	14
Character Mapping.....	15
Marshaling Individual Parameters.....	16
Calling the Windows APIs	17
Structures and Pointers.....	19
Combining Unsafe Code	20
Marshaling Mutable Strings.....	21
Marshaling Delegates	23
Using Reverse P/Invoke.....	24
Marshaling Delegates – Caution.....	25
Generating Signatures	27
Lab: Enumerate Windows	28
P/Invoke Summary.....	29
COM Interoperability	30
COM Interoperability - Challenges.....	31
COM Objects From Visual Studio	32

7-2 | Module 07 - Interoperability

Non-Standard Mappings	33
Manual Customization.....	34
Primary Interop Assemblies	35
Reflection and IDispatch	36
Lifetime Management.....	37
Error Handling	38
Threading Models	39
.NET Objects as COM Components	41
Visual Studio Integration	42
C++ Clients	43
Alternatives for Registration	44
Exposing Interfaces.....	45
Limitations	46
Lab: Dynamic Dispatch Regex Wrapper.....	48
COM Interoperability Summary.....	49
C++/CLI: The Most Powerful .NET Language	50
What Is C++ On The CLR?.....	51
C++ Code Generation.....	52
Basic Class Declaration Syntax	53
More Class Declaration Examples.....	54
Declaring Properties.....	55
Implementing Properties	56
Using Properties.....	57
Delegates and Events	58
Delegates and Events (contd.).....	59
Delegates and Events (contd.).....	60
Virtual Functions	61
Storage And Pointer Model.....	64
Pointers and Handles	65
Boxing (Value Types).....	66

Marshaling (Interop)	68
Marshaling Framework.....	69
CLR Types in the Native World.....	70
Native Types In The CLR.....	71
Uniform Destruction/Finalization	73
Practical Interop Scenarios	75
Lab: Native FileSystemWatcher Low-Fragmentation Heap Wrapper.....	76
C++/CLI Summary	78
Interoperability Considerations	79
CLR Hosting.....	80
CLR Hosting From 10,000 ft	81
Summary	82

7-4 | Module 07 - Interoperability



In This Chapter

- ⌚ Platform Invoke
- ⌚ COM Interop
- ⌚ C++/CLI
- ⌚ Overview of CLR Hosting



Overview of Interoperability

- .NET was designed for interoperability
- Challenges Faced:
 - Locating the other side
 - Passing data correctly (marshaling)
 - Lifetime management (GC vs. ...)

From the first days of the .NET framework, its advent was only possible through interoperability with other technologies. The .NET class library is based on Win32 and many types are thin wrappers on top of existing functionality implemented as part of Windows. Interoperability with system libraries, COM components and C++ code was a primary objective in the design of .NET and the CLR.

The primary challenges in interoperating with unmanaged code have to do with:

- Location – where is the other side? How can I find the other side? How can the other side find me?
- Marshaling – how do I pass the data to the other side? (Some types are easy, some other are not.)
- Lifetime – how do I manage the lifetime of the other side? How does the other side see my lifetime?

Addressing these questions is fundamental to a good interoperability story, and all the .NET interoperability mechanisms we will examine in this module will address them a little bit differently and with varying degrees of freedom. For example, COM interop puts significant limits on marshaling and has a well-defined notion of object lifetime, while P/Invoke allows for more marshaling customization and does not have any notion of lifetime.

The following slides provide a brief overview of the three interoperability mechanisms that will be discussed throughout this module.

Platform Invoke

- Managed code calls into native DLL
- Native code calls into managed code

- Very simple
- C-style exported functions only
- Partial control over marshaling

Platform Invoke (a.k.a. P/Invoke) is the simplest interoperability mechanism, used to great extent by the .NET class library itself. It is a simple attribute-based interoperability approach whereas the signatures of C-style functions in a native DLL are repeated in managed code so that they can be called directly from a managed language. The reverse (calling from native code to managed code) is also possible using delegates (callbacks).

P/Invoke is extremely simple to use, and allows some control over parameter marshaling, making it a good choice for simple API wrappers on top of native C-style DLLs. However, it is limited to C-style (`extern "C"`) exported functions only. (So for example, although theoretically possible, it's highly unwieldy to use P/Invoke for accessing an exported C++ class.)

COM Interoperability

- Managed code calls into COM component
- Native code calls into managed code as if it were a COM component
- Standard COM interfaces
- Very limited control over marshaling
- Locating COM components

COM interoperability allows for two-way communication between COM servers, COM clients and .NET types. Any COM object is accessible from .NET using a layer of interoperability called the Runtime Callable Wrapper (RCW), and any .NET type can be exposed to COM clients using a layer called COM Callable Wrapper (CCW).

COM interop uses standard COM interfaces in both directions, and offers very limited control over marshaling (almost every simple change involves roundtripping the wrappers to IL and back). Locating COM objects is made simple in both directions thanks to the use of the Windows registry.

C++/CLI

- Managed code calls into C or C++ code
- C or C++ code calls into managed code
- **The most powerful .NET language**
- Full control over marshaling
- Fine-grained choice of “what’s managed”
- Mixed managed and native assemblies
- Significantly more complicated

C++/CLI is the ultimate interoperability solution, the Swiss Army knife of interop. It is not simply another mechanism – it’s an entire .NET language in and of itself. C++/CLI allows native code to call into managed code and managed code to call into native code, with almost no limitations whatsoever. Full control over marshaling and very fine-grained control over what code is compiled to native and what code is compiled to IL in mixed images provides a very powerful solution. While much more complicated to use than P/Invoke and COM interop, C++/CLI offers a lot to learn as a language and an interoperability framework.

7-10 | Module 07 - Interoperability

P/Invoke

① Managed code calling native code

- ➊ Custom C-style DLL
- ➋ Windows API (Win32)

```
//Native signature:  
int IsPrime(int number);  
  
//Managed signature:  
[DllImport("MyDll.dll")]  
static extern bool IsPrime(int number);
```

A large, stylized, three-dimensional text logo where the letters 'C', 'O', 'D', 'E' are stacked vertically.

Our first tour into interop-land begins with P/Invoke. As mentioned before, P/Invoke is an interop solution which allows managed code to call into C-style DLLs, invoking any exported functions present in these DLLs. For every method signature in the DLL, a matching signature must be produced in native code to satisfy the compiler and runtime that the interop call can be performed.

In this slide, you see an example of a native method exported from a DLL called `IsPrime`, which takes one integer parameter and returns an integer indicating whether the number is prime or not (0 for composite, 1 for prime). The managed signature corresponding to this method is also shown in the slide – it is an `extern` method decorated by the `[DllImport]` attribute. This method signature allows C# code to access the functionality embedded into the DLL directly, as if it were yet another C# method.

Behind the Scenes

- `extern` satisfies the compiler
- `[DllImport]` satisfies the runtime
 - The DLL is located
 - The entry point is located
 - Parameters are converted
 - Return values are converted back

The `extern` keyword satisfies the C# compiler, which must be able to understand why no method definition was provided for the `IsPrime` method. However, the compiler doesn't play any role in the interoperability story – it is the CLR that performs the managed-to-native transition (and back). The CLR knows about the method because it's decorated with the `[DllImport]` attribute, providing (at the very least) the DLL's location.

At runtime, when the method is called, its DLL is located (lazily – so the DLL is not loaded if the method is not used), the entry point to the DLL called `IsPrime` is located, the parameters are converted if necessary and the return value is converted back after making the call. In this case, we already see a simple example of a type mismatch because the `IsPrime` method originally returns an `int` but the managed signature expects a `boolean` return value. This happens to work because the P/Invoke layer is smart enough to handle this case, but oftentimes we are required to customize parameter and return value marshaling manually.



See the **UsingPInvoke** project in the **Module07_Interop** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how a simple function exported from a C DLL can be called from managed code using the `[DllImport]` attribute and P/Invoke.

Marshaling

- Marshaling: translating parameters
- *Blittable* types have the same representation
 - System.Int32 and C int
 - System.Single and C float
- Some types **do not!**
 - Strings, booleans, arrays vs. pointers

The process of ensuring that parameters are properly converted to their native representation and the return values (including ref and out parameters) are properly converted to their managed representation is called *marshaling*. It is the process of translating information from one environment to the other – in our case, the managed and native worlds.

Some types are fairly easy to translate in this fashion – for example, a System.Int32 (a C# int) has the same representation as a C int variable. However, some other types do not exhibit this behavior – a boolean, a string, an array all have a different managed representation than their native one. For example, a C string is usually ANSI while .NET strings are always Unicode; passing a Unicode string to a method which expects an ANSI string is not going to do much good.

Standard Mappings

- Standard mappings are performed by P/Invoke
 - byte → unsigned char
 - long → __int64
 - bool → int
 - string → char* or wchar_t*

The standard parameter mappings are performed by P/Invoke and do not require any cooperation from the programmer. Some of the most common mappings are outlined on the slide. Even strings can be converted automatically provided that the appropriate information is present.

Character Mapping

- ① Can be performed on per-function basis

```
//Native signature:  
void wputs(wchar_t* s);  
  
//Managed signature:  
[DllImport("MyDll.dll", CharSet=CharSet.Unicode)]  
static extern void wputs(string s);
```



P/Invoke must know what the character set of the native function is in order to perform mapping of characters and strings of characters appropriately. For example, on the slide there's a native signature of a function that takes a `wchar_t*` parameter. Its managed signature takes a string and is decorated with `CharSet.Unicode` to let P/Invoke know that the .NET string should be marshaled as a Unicode string.

In this case, marshaling the string as a Unicode string has a significant performance benefit (other than just correctness) because the string does not have to be converted to ANSI. A direct pointer to the managed string object's underlying character buffer is passed to the native function.

Other valid values for the `CharSet` enumeration include `Ansi` and `Auto`. The latter depends on the platform – if the platform is inherently Unicode (the entire Windows NT family) then the value of `Auto` is `Unicode`; otherwise, it is `Ansi`.

Marshaling Individual Parameters

Customization with [MarshalAs]

```
//Native signature:  
BOOL IsValid(LPCWSTR lpszText);  
  
//Managed signature:  
[DllImport("MyDll.dll")]  
[return: MarshalAs(UnmanagedType.Bool)]  
static extern bool IsValid(  
    [MarshalAs(UnmanagedType.LPWStr)] string text);
```

CODE

An even greater degree of flexibility is enabled by the [MarshalAs] attribute which can be placed on structure fields, method return values and method parameters. This attribute indicates exactly how a specific type should be marshaled by P/Invoke at runtime. In this example, the IsValid method has a return type of BOOL which is in fact an unsigned int – but it's automatically marshaled as a System.Boolean thanks to the [MarshalAs] attribute. In a similar fashion, the string parameter is marshaled as wchar_t* which is what the native method expects.

The UnmanagedType enumeration includes all common Win32 API types as well as COM types such as VARIANT_BOOL.

Calling the Windows APIs

- ⌚ Most Win32 APIs have two versions:

- ⌚ **CreateFileA – ANSI string**
- ⌚ **CreateFileW – Unicode string**

- Use CharSet.Auto with these functions!

When using P/Invoke to call Windows APIs, there is another thing to look out for with regard to string conversions. Almost all Win32 APIs which take a string parameter have two versions – an ANSI version with the A suffix and a Unicode version with the U suffix. Internally, Unicode platforms use only the Unicode version, and the ANSI version is a simple proxy which converts the parameters to Unicode and then proceeds to call the Unicode version.

When using Win32 APIs, you should use the CharSet.Auto enumeration value to ensure that your code remains platform agnostic. Even if you do not care about running on ANSI and Unicode platforms at the same time, using CharSet.Ansi on a Unicode platform is extremely expensive because the .NET string will be converted to ANSI and then converted back to Unicode by the ANSI version of the Win32 API.

7-18 | Module 07 - Interoperability

- C# structs will work
- Pointers are either:
 - Arrays, -or-
 - ref / out parameters

When working with structures, you must duplicate the native definition of the structure in your managed code. The fields of the structure must appear in the same order, and the structure should be decorated with the [StructLayout(LayoutKind.Sequential)] attribute. .NET supports union-like structures with LayoutKind.Explicit and the [FieldOffset] attribute, but these are outside the scope of this module.

When passing structures around in native code, you would usually see a pointer to the structure instead of the structure itself to minimize copying. The .NET equivalent of a pointer to a structure is a ref or out parameter. However, do not confuse pointers which are in fact a ref/out parameter with pointers which are in fact an array! If the underlying API expects an array, then use an array – it will be automatically pinned and passed as a pointer to its first element.

Structures and Pointers

```
//Native signature:  
void Find(CONST ITEM* items, DWORD count,  
          ITEM* lookup, DWORD* index);  
  
//Managed signature:  
[DllImport("MyDll.dll")]  
static extern void Find(  
    ITEM[] items, int count,  
    ref ITEM lookup, out int index);
```

CODE

This example demonstrates how pointers could be the beginning of an array or an individual element. The items parameter is in fact an array of size count which the function expects; the lookup parameter, on the other hand, is a pointer to a single structure. The marshaling directives also differ – the items parameter is a simple array of items, while the lookup parameter is passed with the ref keyword.

MCT USE ONLY. STUDENT USE PROHIBITED

Combining Unsafe Code

• Alternatively, pointers can be used

• In unsafe context, requiring fixed

```
unsafe
{
    fixed(ITEM* p = &items[2])
    fixed(ITEM* q = &lookup)
    {
        Find(p, items.Length-2,
              q, out index);
    }
}
```

CODE

An alternative to using arrays and ref/out parameters is using unsafe code and pointers. The syntax of unsafe code and the fixed statement is outside the scope of this module, but the example on the slide still demonstrates how pointers can be used instead of arrays and individual elements in the previous example. Furthermore, in this code example, a pointer to the 3rd element of the array is passed to the native API, enabling an additional degree of control and customization over marshaling.

Marshaling Mutable Strings

- ① If a string requires modification, use `StringBuilder` with a capacity

```
//Native signature:  
void FillString(char* s, char fill);  
  
//Managed signature:  
[DllImport("MyDll.dll")]  
static extern void FillString(  
    StringBuilder text, char fill);
```

CODE

.NET strings are immutable. If a managed string is passed to native code, the unmanaged side is not allowed to modify the string. (This can have disastrous consequences on other users of the string, especially if the string is interned.)

If you need to pass a mutable string to unmanaged code, use a `StringBuilder` pre-initialized with some characters or capacity. Even if you accidentally get a solution to work with mutating strings, *never* rely on this kind of solution. Always use `StringBuilder` when interacting with unmanaged code which needs to modify a string.

Marshaling Structures and Strings



See the **UsingPInvoke** project in the **Module07_Interop** solution under the SampleCode folder for more information about this demo in the code comments.
In this demo, you will see how structures, arrays of structures, strings and mutable strings can be marshaled across the managed and unmanaged P/Invoke boundary.

Marshaling Delegates

- Native-to-managed callbacks: delegates

```
//Native signature:  
typedef BOOL (_stdcall *PFNMATCH)(char* text);  
DWORD Find(PFNMATCH pfnMatch);  
  
//Managed signature:  
delegate bool IsMatch(string text);  
[DllImport("MyDll.dll")]  
static extern int Find(IsMatch match);
```

CODE

Callbacks (unmanaged function pointers) can be marshaled using P/Invoke through the use of managed delegates. If you pass a managed delegate across the P/Invoke boundary, the marshaler automatically obtains an unmanaged function pointer for the delegate and passes that through. (You can also obtain the function pointer manually using the Marshal class, but this is outside the scope of this module.)

It's crucial to note that all managed delegates are marshaled with the `_stdcall` calling convention, so the unmanaged side must have the `_stdcall` calling convention when calling through the unmanaged function pointer. Failure to do will corrupt (imbalance) the stack and result in an unpredictable runtime error.

Using Reverse P/Invoke

Passing a delegate (anonymous method):

```
.IsMatch match = delegate(string s) {  
    return s.StartsWith("A");  
};  
int index = Find(match);
```

CODE

Passing a delegate (lambda):

```
.IsMatch match = s => s.StartsWith("A");  
int index = Find(match);
```

The code on this slide demonstrates that an anonymous method or a lambda expression can be used as an unmanaged function pointer. Both are bound to a real delegate instance at runtime, and that instance is converted to an unmanaged function pointer and passed across the P/Invoke boundary.

Marshaling Delegates – Caution

- Keep the delegate alive until it's unused:
 - Store a static/member reference
 - Use GC.KeepAlive
 - Use GCHandle

One scenario which requires great caution is when the unmanaged code keeps the unmanaged function pointer value for future use. In this case, it's crucial to prolong the lifetime of the managed delegate instance for as long as the unmanaged code prolongs the lifetime of the unmanaged function pointer. Failure to do so might cause the delegate to be garbage collected, and subsequent uses of the unmanaged function pointer will call into unknown memory. This is extremely dangerous.

Keep it simple: If the native side keeps the delegate for later use, **you must keep it alive** on the managed side!

This can be done by extending the lifetime with a static reference, by using the GC.KeepAlive method or by using an explicit GCHandle. One way or another, the managed delegate instance must outlive the unmanaged function pointer under all circumstances.



See the **UsingPInvoke** project in the **Module07_Interop** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how delegates can be marshaled across the P/Invoke boundary and how failure to prolong the lifetime of a delegate instance causes a difficult-to-understand error at runtime when the unmanaged side attempts to call through the unmanaged function pointer.

Generating Signatures

- Remembering signatures is tedious!
- **PInvoke.net** – a collection of signatures online
- **P/Invoke Interop Assistant** – a tool for generating signatures

Generating P/Invoke signatures by hand is error-prone and tedious, especially if you're dealing with a large API base such as the Win32 libraries. Fortunately, automatic tools are available at your disposal: the first is the pinvoke.net web site, which features a large collection of P/Invoke signatures ready for use; the other is the P/Invoke Interop Assistant, an automatic Microsoft utility which generates P/Invoke signatures from unmanaged code snippets (header files).

<http://msdn.microsoft.com/en-us/magazine/cc164193.aspx>

Lab: Enumerate Windows



Exercise 1 – P/Invoke and Reverse P/Invoke:

Use P/Invoke to print the window titles of all windows on your system. You are free to implement your solution as a console application or a UI application.

To enumerate all windows, use the Win32 **EnumWindows** API. It accepts a function pointer as a callback which you will need to marshal as a delegate.

To obtain the window's title provided you have its handle, use the Win32 **GetWindowText** API. It fills a string which you will need to marshal as a **StringBuilder**.

You can find the solution for this lab in the **InteropExerciseSolution** solution under the **ExerciseSolutions\Module07_Interop** folder (specifically the **PInvoke** project).

P/Invoke Summary

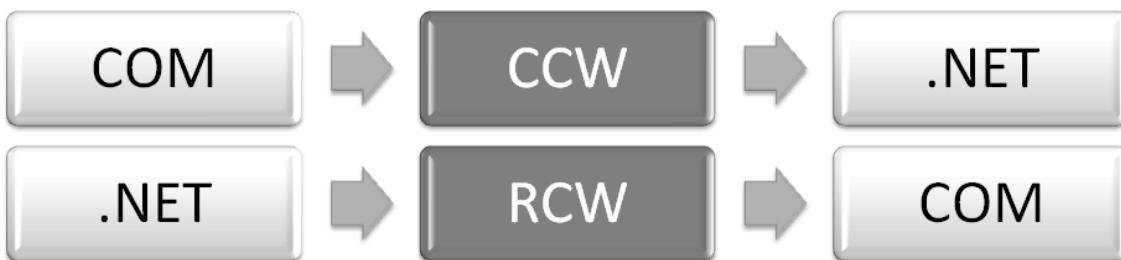
- Easy automatic location and marshaling
- Customizing marshaling for strings, arrays, structures and pointers
- Works with C-style exported DLL functions only



This concludes the first part of this module, dealing with P/Invoke interoperability. We have seen that P/Invoke offers relatively easy and customizable location and marshaling services, and that P/Invoke works with C-style exported functions from DLLs only.

COM Interoperability

- Managed code calling COM objects
- Native code calling .NET components exposed as COM objects
- Interoperability enabled by runtime wrappers



Our next interoperability mechanism is COM-based interop, which allows two-way communication between COM objects and managed objects. Dynamic runtime wrappers create the abstraction for .NET code calling into COM libraries and for COM clients calling into .NET code “hiding” behind a COM façade.

The two types of wrappers are the COM Callable Wrapper (CCW) which enables COM clients to interact with .NET types as if they were COM objects, and the Runtime Callable Wrapper (RCW) which enables .NET code to interact with COM objects as if they were .NET types.

COM Interoperability - Challenges

- COM defines more rules than C
 - Error handling (HRESULT, IErrorInfo)
 - Life-time management (reference counting)
 - Threading model (apartments)
- And of course, there's marshaling!

Working with COM interop is often difficult because, much like the CLR, the COM infrastructure defines many rules for components and objects implemented on top of it. Error handling, life-time management and the threading model are examples of these rules, and they differ greatly from the rules of CLR objects.

We will see the differences between COM and CLR rules later in this module, as they become relevant for interop purposes.

COM Objects From Visual Studio

- Project → Add Reference → COM
- An *interop assembly* is generated
- Alternative: Use the *tlbimp.exe* tool

Using a COM object from .NET code is as easy as one-two-three with Visual Studio. Much like a managed reference can be added to a managed project, a COM reference can be added using the References dialog (the COM tab). When a COM type library is selected from the list (which is prepopulated with values from the registry), an *interop assembly* is generated for that type library.

The interop assembly contains metadata (no actual IL instructions) which allow the CLR to generate an RCW at runtime, enabling managed code to seamlessly call into COM objects. The wrapping appears perfect, and the steps to take in order to interact with a COM object are no different from the steps to take to interact with a regular managed type:

- Create an instance
- Call methods
- Register to events

When generating the interop assembly, Visual Studio performs several mappings which reflect the differences and similarities between the COM and CLR worlds:

- HRESULTs are mapped to CLR exceptions
- Properties are mapped to CLR properties
- IDL structures, enums and interfaces are mapped to their CLR counterparts

Visual Studio's work can be replaced by manually executing the *tlbimp.exe* command-line utility, which creates an interop assembly from a COM type library file. There are some minor customizations which *tlbimp.exe* can perform and Visual Studio can't, but most projects will work just fine with the Visual Studio-generated interop assembly.

Non-Standard Mappings

- Sometimes the generated interop assembly is wrong!
- The only way to customize marshaling is by manually editing the interop assembly

When the generated interop assembly has wrong marshaling information, there is nothing to do but to manually edit the interop assembly. While very tedious, this is the only way to truly customize the marshaling process of COM interop.

Examples of things that can go wrong in the interop assembly include:

- Arrays mapped to a single ref parameter
- Success **HRESULTs** eliminated (**S_OK** vs. **S_FALSE**)

Manual Customization

- Disassemble:

```
ildasm Interop.MyCOM.dll /out:Interop.MyCOM.il
```

- Perform modifications, e.g.:
 - int32& → int32[] marshal([])
- Reassemble:

```
ilasm /dll Interop.MyCOM.il /resource:Interop.MyCOM.res
```

This slide lists the steps required to manually customize the interop assembly. First you must disassemble it into IL, then edit the IL manually, and finally reassemble it back into an interop assembly which your managed code can then reference.

Primary Interop Assemblies

- Signed interop assembly supplied by the component's vendor
 - Marked with [PrimaryInteropAssembly] attribute
- Create with *tlbimp.exe /primary*

COM interop includes the trusted concept of a primary interop assembly, which is a signed interop assembly supplied by the vendor of the COM component. There is nothing special about a PIA except that it is marked with the [PrimaryInteropAssembly] attribute, has a strong name (signed) and is usually installed to the GAC by the COM object's installer. Visual Studio checks the GAC for PIAs before generating an interop assembly, so if there is a PIA on your system for a component you happen to use – that PIA will be used when adding a reference to the component in Visual Studio.

It is advised to always use the PIA for a COM object instead of relying on your own interop assembly, because assumedly PIAs undergo an implementation process by the component vendor who has the most familiarity with the actual component. If you are a COM component vendor, then you should consider creating your own PIAs using the tlbimp.exe utility with the /primary flag.

Reflection and IDispatch

- What if there's no type library?
- What if I'm referencing COM objects dynamically?
- Use Reflection – it will use IDispatch!
 - `Type.GetTypeFromProgID`
 - `Activator.CreateInstance`
 - `Type.InvokeMember`

Just as there are scenarios where you need to use a .NET type dynamically at runtime, oftentimes you need to bind to a COM object at runtime. (Incidentally, scripting environments and languages such as Visual Basic 6 have no other way of binding to COM objects.)

Binding dynamically to a COM object requires the use of the IDispatch interface. Fortunately, .NET Reflection abstracts away the difficulties of using IDispatch, and provide a clean interface for interacting with COM objects. The `Type.GetTypeFromProgId` method retrieves a COM type from its human-readable ProgId, the `Activator.CreateInstance` method can be used to dynamically instantiate that type, and the `Type.InvokeMember` method (along with others) can be used to dynamically execute operations on the COM object.

Note that as any use of Reflection, the use of IDispatch is significantly more expensive than direct calls. Avoid this price if possible by interacting directly with the custom interface implemented by the COM object, and not IDispatch.

Lifetime Management

- .NET objects are subject to **GC**
- COM objects are subject to **RC**
- When the RCW is collected, it releases the COM object
 - Marshal.ReleaseComObject
 - Marshal.FinalReleaseComObject
- **Beware of RCW-CCW cycles!**

The lifetime management mechanisms of COM and .NET objects are extremely different, although both are subject to a form of garbage collection. .NET objects are collected by the .NET tracing GC (see module 2), while COM objects are collected using reference counting. Because there is no reference count embedded in a standard .NET object, the RCW created to wrap the COM object explicitly calls Release when it is finalized. Waiting for the RCW to be finalized creates a potential memory pressure scenario, and therefore it is recommended to deterministically dispose of COM objects by using the Marshal.ReleaseComObject or Marshal.FinalReleaseComObject API calls.

Another potentially dangerous scenario is a cycle between COM and .NET objects referencing one another (e.g. through the use of COM events). In this kind of scenario, there is a memory leak because there is no way for the garbage collector to know that a COM object is keeping a managed object alive. Cycles of this sort must be manually broken by using the Marshal class APIs.

Error Handling

- Failure HRESULTs are converted to CLR exceptions
- Success HRESULTs are not mapped

Error handling is another area where there is a significant difference between COM and .NET. COM objects report errors using error codes known as HRESULTs, each associated with a meaning. .NET code throws exceptions when an exceptional condition occurs. COM interop bridges the two worlds by converting COM HRESULTs to managed exceptions. Known HRESULTs will be converted to known exceptions, for example `E_INVALIDARG` will surface as an `ArgumentException` instance. Custom HRESULTs are mapped to `COMException` instances with the `ErrorCode` property which can be used to detect the specific HRESULT.

Success HRESULTs are not mapped by default, and so if there is a difference between success HRESULTs that a method might return (e.g. `S_FALSE` vs. `S_OK`), you must modify the interop assembly and specify the `preservesig` metadata attribute on the method to get the HRESULT instead of the automatic HRESULT-to-exception conversion performed by the RCW.

Threading Models

- COM objects can specify an apartment requirement
- A .NET thread has a default apartment mode of MTA
 - Thread.ApartmentState
 - [STAThread], [MTAThread]

Lastly, COM objects have a notion of a threading model which affects the way threads access the object. COM objects live in an “apartment” which can be the single-threaded apartment or the multi-threaded apartment:

- MTA – multiple threads can access the object
- STA – one **specific** thread can access the object

Cross-apartment calls are made through proxy-stub pairs (expensive!), and is best avoided. .NET threads have an apartment mode of MTA by default, and this can be modified in case a COM object requires STA (especially UI objects tend to require STA). This can be done by setting the Thread.ApartmentState property *before* the thread is started with Thread.Start, or by placing the [STAThread] or [MTAThread] attribute on the thread’s entry point. (This is the only way to affect the apartment state of the application’s Main method.)



Accessing COM Objects from .NET

See the **CalculatorClient** and **SimpleCOMCalculator** projects in the **Module07_Interop** solution under the SampleCode folder for more information about this demo in the code comments. In this demo, you will see how a simple COM object is accessed from managed code. Using the COM object from C# is as easy as using any other .NET type, thanks to the interop assembly created by Visual Studio and the RCW created at runtime by the CLR.

.NET Objects as COM Components

- .NET objects exposed as COM components can be accessed from almost any Windows language!
- A type library is generated from the assembly
 - Registered under HKCR
- The .NET assembly is not in the registry!

The other direction of COM interop enables .NET types to be accessible from almost any Windows language and scripting environment by exposing them through a COM façade. Every .NET assembly can have a type library generated from it, and can be registered under the HKCR (HKEY_CLASSES_ROOT) key in the Windows registry. Note that the path to the assembly itself is *not* placed in the registry – the HKCR entry points to mscoree.dll, which acts as a shim for instantiating the .NET type and transferring control to the CLR. The actual .NET assembly must still be loaded by the usual .NET binding rules – either from the caller's private path or from the GAC.

Visual Studio Integration

- Project → Properties → Build → Register for COM Interop
- In *AssemblyInfo.cs*, use
[assembly:ComVisible(true)] -or-
- Specify [ComVisible(true)] for individual types, interfaces, methods etc.

Visual Studio can take care of assembly registration automatically – you can take advantage of this option by using the Build tab on the project properties tab sheets, and checking the Register for COM Interop checkbox. However, it's not enough for the assembly and its types to be visible by COM – you must make sure that either your AssemblyInfo.cs file contains an assembly-level attribute indicating that your assembly's types are visible to COM – [assembly: ComVisible(true)], or your individual types and interfaces must be decorated with the [ComVisible(true)] attribute.

C++ Clients

- Clients can use the type library as usual:

```
#import "MyNetObject.tlb" no_namespace  
  
IMyNetClassPtr p(CLSID_MyNetClass);  
p->MyMethod();  
p->MyProperty = 5;
```

CODE

C++ clients can access the .NET type library as if it were yet another COM object. In this example, a C++ client using the com_ptr wrapper mechanism accesses a COM object which is in fact backed by a .NET assembly. There is no visible difference as far as the client is concerned, thanks to COM's location and language independence.

Alternatives for Registration

- regasm
- tlbexp

Alternatives to Visual Studio automatic registration include the regasm and tlbexp tools. These tools are intended for advanced scenarios and are outside the scope of this module.

REGASM .EXE

- /tlb – generates a type library as well
- /codebase – instead of GAC or private path (for development purposes)

TLBEXP .EXE

- Generates a type library
- Does **not** register the assembly
- Prefer **REGASM .EXE**

Exposing Interfaces

- Expose explicit interfaces
- [ClassInterface]

When exposing a .NET type to COM clients, remember that COM clients expect COM objects to adhere by COM rules, namely that interfaces are immutable. While .NET clients have no problem using a modified interface, COM clients will require a recompilation in order to see the changes to the interface.

Prefer exposing an explicit interface from your .NET objects and versioning the interfaces under COM rules if your .NET objects are exposed to COM clients. Your choice of exposing an interface to clients relies on the [ClassInterface] attribute placed on your .NET class, which takes one of the following values from the ClassInterfaceType enumeration:

ClassInterfaceType.AutoDispatch

Default interface for the class (_ClassName), **IDispatch**-based invocation

ClassInterfaceType.AutoDual

Default dual interface for the class (_ClassName)

ClassInterfaceType.None

Explicit interfaces only (recommended)

Limitations

- Customizing marshaling is impossible
- Static members can't be exposed
- Overloaded methods can't be exposed
- Exceptions are reported as HRESULTs

CCW interoperability (from COM clients to .NET objects) has certain limitations. It's impossible to customize marshaling, there are limitations on the types and methods that can be exposed, rich .NET exceptions are reported as HRESULTs and so on. There is a boundary between the COM and .NET worlds, and crossing it often requires compromise. Other advanced topics which are outside the scope of this module include:

- Non-standard creation of COM servers
- DCOM integration
- Registration-free COM integration
- ActiveX control integration
- Consuming and exposing events

Recommended reading: .NET and COM – The Complete Interoperability Guide (Adam Nathan)

Accessing .NET Types from COM



See the **NativeClient** and **UsefulManagedLibrary** projects in the **Module07_Interop** solution under the **SampleCode** folder for more information about this demo in the code comments. In this demo, you will see how a .NET type can be accessed by a COM client through the use of the CCW façade. The native client is completely unaware of the fact it is talking to a .NET type on the other side of the interop boundary.

Lab: Dynamic Dispatch Regex Wrapper



Exercise 2 – COM Interoperability (Optional):

Use Reflection (**Type.GetTypeFromProgID** etc. as outlined in the slides) to instantiate the *MSCAL.Calendar.7* COM object and query it for the **Year**, **Month** and **Day** properties (of type short).

Write a .NET class that wraps the .NET regular expression facilities (in *System.Text.RegularExpressions*) by providing a single **Match** function with the following signature: **bool Match(string regex, string text)**. Register the assembly for COM interop and consume it from a C++ or VBScript client.

You can find the solution for this lab in the **InteropExerciseSolution** solution under the **ExerciseSolutions\Module07_Interop** folder (specifically the **COMWithReflection**, **RegexWrapper** and **RegexVBSClient** projects).

COM Interoperability Summary

- Simple marshaling model
 - Hardly any or no customizations
- Overcoming the differences:
 - Registration model
 - Lifetime management
 - Error handling
 - Threading model



In the COM interop part of this module, we have seen the simple interoperability model enabled by CCWs and RCWs in both directions of the interop boundary. We have also seen how the unfortunate differences between the CLR and COM worlds (including location, lifetime, error handling and threading) can be overcome.

C++/CLI: The Most Powerful .NET Language

- Deterministic Cleanup
 - Templates
 - Native Types
 - Multiple Inheritance
 - STL
 - Generic Algorithm
 - Pointers
 - Copy Constructor
 - Assignment Operator
 - Legacy Code
- 
- GC, Finalizer
 - Generics
 - Reference & Value Types
 - Interfaces
 - Safe, Verifiable
 - Security
 - Properties
 - Delegates, Events
 - .NET Framework

Our final interoperability mechanism is C++/CLI, which is not just an interoperability mechanism – it is an entire .NET language. In this module, we refer to it as the most powerful .NET language, because it bridges the worlds of ISO C++ and the CLR constructs as well as the .NET framework. This slide summarizes some of the apparently irresolvable differences between the native C++ and CLR worlds – and yet C++/CLI is a single unified language based on C++ which enables access to every and all CLR constructs.

What Is C++ On The CLR?

- ISO standard C++ on the CLR
- Language binding to .NET framework
- Seamless managed and native interop

C++/CLI is a first-class managed language with excellent syntax mapping for all CLR and language features. However, C++/CLI is also standard ISO C++ - and it's possible to compile every corner of C++ syntax using C++/CLI, down to IL or native code. The use of C++/CLI binds C and C++ solutions to the .NET framework, enabling them to directly access managed code without the use of additional custom interop boundaries.

The interoperability story of C++/CLI is so good and seamless that it has been termed "It Just Works" (IJW). The creation of mixed native and managed images enables truly transparent interoperability, where two methods from the same source file might lie on different sides of the interoperability boundary.

C++ Code Generation

- Compiles to MSIL
 - /CLR – Mixed Mode Images (native and MSIL)
 - /CLR:Pure – MSIL Only
 - /CLR:Safe – Verifiable MSIL
 - /CLR:oldSyntax – Managed C++
- Compiles to Native

There are multiple ways to compile C++ code starting in Visual Studio 2005. While C++ code can still be compiled to native machine code (x86, x64 and ia64 are currently supported), C++ can also be compiled to IL in whole or in part.

The /CLR switch produces mixed mode images, which contain native machine code as well as IL code. Specifically, those sections of the code that are marked with #pragma unmanaged are compiled to native code, and sections of the code marked with #pragma managed (or files compiled with /CLR) are compiled to IL. This means that the entire C++ standard, with very few exceptions, can be compiled to IL and executed naturally on the .NET platform.

The /CLR:Pure switch ensures that the entire assembly is compiled to IL, and that there are no C++-specific constructs that can't be translated to IL. There are only very few such constructs, e.g. inline assembler.

The /CLR:Safe switch ensures that the produced assembly contains only verifiable IL. This means that pointers, multiple inheritance and many other C++-only idioms are not supported. This is a rather esoteric switch – using it means that you're writing C# code in C++/CLI.

Finally, to support Visual Studio 2003 (which shipped with the Managed Extensions for C++, the predecessor of C++/CLI), there is also a /CLR:oldSyntax switch which understands the old MC++ syntax.

Basic Class Declaration Syntax

- ① Types are declared “adjective class”:
- ② Fundamental types are mapped to each other

```
class CNative  
ref class CManaged  
value class CValue  
interface class IInterface  
enum class E
```

The word "CODE" is written in a large, bold, white font with a black shadow effect.

Types in C++/CLI are declared very similarly to C++ - the class keyword declares a class, and special adjectives attached to it declare specific other types. The class keyword alone declares a native C++ class which can be compiled to MSIL, but cannot be consumed by other .NET languages. The ref class keyword pair declares a reference type (which must compile to MSIL and can be consumed by other .NET languages). The value class, interface class and enum class declarations are similar in nature – they declare a value type, interface and managed enum, respectively, which can be consumed by other .NET languages. The public accessibility modifier can also be appended to the latter four declarations.

Finally, fundamental types are mapped to each other for convenience. For example, there's no need to perform conversions between a C++ int and a .NET Int32.

More Class Declaration Examples

- ① Extending the ISO C++ language:
- ① The same syntax for native and managed types

```
class CShape abstract ...
class CNative sealed ...
class CDerived : public CNative {};
//Error: CNative is sealed
```

A stylized, three-dimensional text logo for "CODE". The letters are white with a black shadow effect, giving it a metallic or plastic appearance.

Some keywords can be placed after the class name, declaring the intent of how this class can be used rather than what this class is. This is an extension of C++ which applies to purely native code as well (compiled without /CLR), and entails of the ability to declare that a class be abstract (can't be instantiated, must be inherited), sealed (can't be inherited), abstract sealed (a rough equivalent of the C# static class) etc.

Declaring Properties

```
1 value class Complex
2 {
3     public:
4         property double Real;
5         property double Imaginary;
6         property double R
7         {
8             double get();
9             void set(double newR);
10        }
11        property double Theta
12        {
13            double get();
14            void set(double newTheta);
15        }
16    };
```

A large, stylized, three-dimensional white text logo with a black shadow effect, spelling the word "CODE".

Properties can be declared using the automatic property syntax, shown in lines 4-5. (Note how the C# 3.0 automatic property syntax mirrors this to a great extent.) Properties declared in this automatic manner receive a backing field and an accessor pair (get, set) automatically. Explicit properties can be declared using the syntax shown on lines 6-10 and 11-15. After declaring the property, accessors must be explicitly declared with their return type and parameter list. The implementation of the properties is not shown here – envision this as a the header file for the Complex value class (C# struct).

Implementing Properties

```
1 double Complex::R::get() {  
2     return Math::Sqrt(Real*Real + Imaginary*Imaginary);  
3 }  
4 void Complex::R::set(double newR) {  
5     Real = Math::Cos(Theta) * newR;  
6     Imaginary = Math::Sin(Theta) * newR;  
7 }  
8 double Complex::Theta::get() {  
9     return Math::Atan2(Imaginary, Real);  
10 }  
11 void Complex::Theta::set(double newTheta) {  
12     Real = Math::Cos(newTheta) * Real;  
13     Imaginary = Math::Sin(newTheta) * Real;  
14 }
```

CODE

This is an implementation of the R and Theta properties shown in the previous slide. This code belongs in a C++ source (.cpp) file, or after the class declaration in the header file. The property name and accessor name are required to implement the property. Properties can refer to other properties using their name, without the *this* qualifier.

Using Properties

```
1 Complex complex(2.0, 3.0);
2 Console::WriteLine("{0} + {1}i",
3     complex.Real, complex.Imaginary);
4 Console::WriteLine(
5     "= {0:F2}*(cos({1:F2}) + i*sin({1:F2}))",
6     complex.R, complex.Theta);

– //Output:
– 2 + 3i
– = 3.61*(cos(0.98) + i*sin(0.98))
```

CODE

When used outside the type, properties behave like fields. In lines 3 and 6 the Real, Imaginary, R and Theta properties are accessed, and used directly as if they were fields. The compiler ensures that the appropriate get and set accessors are called when necessary.

Delegates and Events

```
1 ref class ChatClient
2 {
3     private:
4         void OnMessageArrived(
5             System::Object^ sender,
6             MessageArrivedEventArgs^ args);
7         System::String^ _name;
8         ChatServer^ _server;
9     public:
10        ChatClient(System::String^ name,
11                    ChatServer^ server);
12        void SendMessage(System::String^ message);
13    };
```

The word "CODE" in a large, bold, sans-serif font, with each letter having a thick, three-dimensional appearance.

Delegates and events are declared in a similar fashion to the C# delegates. In this slide and the following two slides, we will see an example of implementing a chat system with a client and a server using delegates and events.

First, the client class declares a private function called `OnMessageArrived` which handles messages received from the server. This is the function that will register for an event exposed by the server. Next, we declare a constructor that takes a client name (user name in the chatroom) and a reference to the server, so that the client can subscribe to events and send messages. Finally, the `SendMessage` function provides the ability to send a message from this client to the rest of the clients (the chatroom).

Delegates and Events (contd.)

```
1 delegate void MessageArrivedEventHandler(
2     System::Object^ sender,
3     MessageArrivedEventArgs^ e);
4
5 ref class ChatServer
6 {
7 public:
8     void SendMessage(
9         System::String^ from,
10        System::String^ message);
11     event MessageArrivedEventHandler^ MessageArrived;
12 };
```

CODE

Next, we declare the delegate type – `MessageArrivedEventHandler` – which takes an object and a `MessageArrivedEventArgs` type. This is the natural .NET way of declaring event handlers – the first parameter is the event source and the second parameter is the event information.

On lines 5-12, the server itself is declared. The server provides a `SendMessage` function which allows clients to send messages to the chatroom, and a `MessageArrived` event which clients register to. This event is invoked whenever a new message arrives, and is handled by clients to display the message.

Delegates and Events (contd.)

```
1 void ChatServer::SendMessage(...)  
2 {  
3     MessageArrived(this,  
4                     gcnew MessageArrivedEventArgs(from, message));  
5 }  
6 ChatClient::ChatClient(...)  
7     : _name(name), _server(server)  
8 {  
9     _server->MessageArrived +=  
10        gcnew MessageArrivedEventHandler(this,  
11                                         &ChatClient::OnMessageArrived);  
12 }  
13 void ChatClient::SendMessage(System::String^ message)  
14 {  
15     _server->SendMessage(_name, message);  
16 }
```



Finally, in the implementation of the ChatServer class, the SendMessage function invokes the MessageArrived event. The invocation syntax is identical to the C# syntax – the name of the event followed by its arguments, in this case the server itself and a new instance of the MessageArrivedEventArgs data structure.

In the implementation of the ChatClient class, specifically in its constructor (lines 6-12) the client registers for the MessageArrived event by providing a new event handler. This event handler is a delegate of the MessageArrivedEventHandler type, and it is initialized (on lines 10-11) with the client's *this* parameter and the OnMessageArrived function pointer. This initialization syntax for managed delegates is different from C#.

As a result, when a message arrives to the server (through ChatClient::SendMessage and then ChatServer::SendMessage), the MessageArrived event is invoked and all clients registered to the MessageArrived event have their OnMessageArrived event handler function invoked to handle the message.

Virtual Functions

```

1 interface class I1 { int f(); int h(); };
2 interface class I2 { int f(); int i(); };
3 interface class I3 { int i(); int j(); };
4
5 ref class R : I1, I2, I3 {
6 public:
7 virtual int e() override; //error, no virtual e()
8 virtual int f() new; //new slot, doesn't override any f
9 virtual int f() sealed; //overrides I1::f and I2::f      • Explicit, multiple and
10 virtual int g() abstract; //same as "=0"
11 virtual int x() = I1::h; //overrides I1::h
12 virtual int y() = I2::i; //overrides I2::I
13 virtual int z() = j, I3::i //overrides I3::j and I3::I
14 };

```

C++/CLI naturally supports interface implementations, overriding methods, sealing them and defining them as abstract. However, there are a few subtle differences if you're converting from C# - namely, C++/CLI supports a more extensible set of functionality than C#.

In this example, lines 1-3 define three interfaces with overlapping method names. The only way to implement these three interfaces in C# using one class would be using explicit interface implementation (specifying the interface name before the method name, as in e.g.

IComparable.CompareTo(...))). However, C++/CLI is not limited in this regard, and lines 11-13 show how virtual methods that don't even have the same name as the interface method can still be bound to the interface methods that they are overriding, provided that they have the appropriate method signature.

In line 13, a single method is actually bound to two different interface methods, and implements them both – this is highly useful if there is overlapping functionality (not only names) in the interfaces you implement.

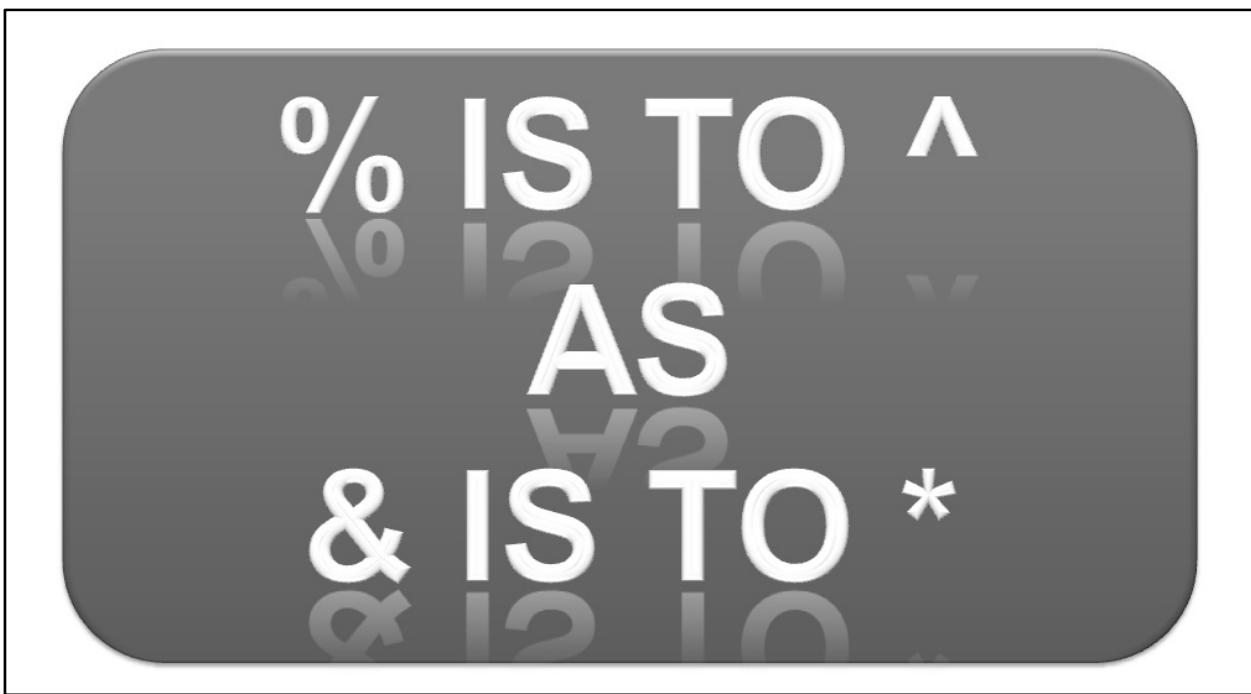
Finally, lines 7-10 demonstrate the use of the override, new, sealed and abstract keywords, applied after the method name and formal parameters. Note that the virtual modifier in the beginning of the declaration is still required, even when using the new keywords.



Fundamental Language Constructs

See the **UsingCppCli** project in the **Module07_Interop** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how to use fundamental language constructs in C++/CLI to declare various kinds of types.



We've already seen several code samples where the `^` symbol was used after the name of a managed type. It's time that we understand what this symbol and the other symbols actually mean with regard to variables and types.

Consider the following example from C++: If you have an integer, then you can declare a pointer to it – an `int* p`. You can then read the value of that pointer using the dereference operator, i.e. `*p`. Finally, you can even declare a reference to the integer, if you have the integer itself or a pointer to it – an `int&`.

These forms of pointers and references are also available in C++/CLI for managed types. Because managed objects are allocated on the garbage-collected (GC) heap, providing direct pointer semantics (including pointer arithmetic and ordering) would be extremely dangerous, especially considering that objects move around when a GC occurs. Therefore, the C++/CLI language provides new symbols to declare a reference (essentially a pointer, also known as a *tracking handle*) to a managed object and to declare a reference to a value type (essentially a pointer, but known in C# as a *ref* or *out* modifier).

If we have a managed type like `System::String`, then an instance of it can be referred to only through a `System::String^` – note the `^` symbol. There is no way to copy the string locally – it's a reference type, and therefore it's always accessed through a tracking handle. On the other hand, an `Int32` can be passed as a reference parameter to a function, in which case the formal parameter type may be `Int32%`. Other, more complex types (such as structs or references) can be addressed with the `%` symbol as well.

Storage And Pointer Model

- On the native heap (native types):

```
T* t1 = new T;
```

- On the GC heap (CLR types):

```
T^ t2 = gcnew T;
```

- On the stack, or as a class member:

```
T t3;
```

There are multiple ways to store information and reference that information through variables in C++/CLI. This slide shows some of these ways. For native types, we can create an instance of the type on the native heap using the standard *new* operator. This produces a standard pointer, using the * symbol. These pointers are completely stable, unaffected by GC and support the entire bag of tricks such as pointer arithmetic, integral conversions of pointers and to pointers, etc. It can be said that * pointers are inherently unsafe.

On the other hand, managed types are created on the GC heap, and therefore they are created with the *gcnew* operator and are assigned back to a tracking handle with the ^ symbol. Calling *delete* on tracking handles is optional – non-deterministic finalization can be used to destroy objects which require cleanup.

Finally, managed and native types can be allocated on the stack or as a class member, and then their lifetime depends on the lifetime of the enclosing method or class instance. Doing so provides a guarantee of deterministic finalization, which will be covered later in this chapter.

Pointers and Handles

```
1 //Pointer to the native heap:  
2 NativeBox* nativeBox = new NativeBox;  
3 nativeBox->Boxify();  
4 (*nativeBox).Boxify();  
5 //Pointer to the managed (GC) heap:  
6 ManagedBox^ managedBox = gcnew ManagedBox;  
7 managedBox->Boxify();  
8 (*managedBox).Boxify();  
9 //error C2440: 'initializing' :  
10 //cannot convert 'cli::interior_ptr<Type>' to 'int *'  
11 //int* pToTheBox = &managedBox->InTheBox;  
12 //Declare a pinning pointer, and then reach  
13 //for the actual address:  
14 pin_ptr<int> pToTheBox = &managedBox->InTheBox;  
15 int* p = pToTheBox;
```

A large, stylized, metallic-looking word "CODE" with a drop shadow effect.

This slide demonstrates some more advanced functionality that is enabled through pointers and tracking handles. In lines 2-4, the NativeBox class is instantiated on the native heap and is used through a NativeBox* - a standard pointer. Dereferencing it and calling methods on the pointer itself is standard C++.

Next, in lines 7-9, the same process is repeated with a ManagedBox class, which is allocated on the GC heap using the *gcnew* operator. Dereferencing it and calling methods on the tracking handle is performed in a very similar way to a standard pointer.

Finally, lines 18-19 demonstrate that it's possible to obtain a pinning pointer (see the Garbage Collector module for more information about pinning) to a member of a managed type, provided that the enclosing instance can be pinned. (An alternative, shown in line 14, which is reaching directly into the managed type and obtaining a pointer fails with a compilation error.) Note that the pinning pointer is freed when the enclosing block completes – this means that the actual pointer should not be stored elsewhere for future use. Use pinning pointers locally and forget about the address immediately after disposing them.

Boxing (Value Types)

```
1 generic <typename T>
2 void Swap(T% first, T% second) {
3     T temp = first;
4     first = second;
5     second = temp;
6 }
7 void BoxingAndUnboxing() {
8     int value = 42;
9     int^ boxed = value;
10    System::Object^ obj = boxed;
11
12    int copy = *boxed;      //Strongly-typed, no cast
13    int% refToTheBox = *boxed;
14    int newValue = 43;
15    Swap(*boxed, newValue);
16 }
```

A large, stylized, three-dimensional text logo where the letters 'C', 'O', 'D', 'E' are stacked vertically with horizontal shadows, giving it a 3D effect.

Boxing is implicit and strongly typed (see lines 9, 10), meaning that it's possible to refer to boxed objects and even look "into the box" without performing actual unboxing. Boxing invokes the copy constructor for types that have a copy constructor.

Unboxing is explicit (like in C#, line 12), and again it's possible to refer inside the box as in line 13. The reference into the box allows us to modify the boxed contents without performing the unboxing operation (which is relatively expensive for large objects). The Swap method call (line 15) demonstrates how we can swap the contents of the box for something else without ever performing unboxing.

Heap, Stack and What's in Between



See the **UsingCppCli** project in the **Module07_Interop** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how to allocate managed types on the GC heap, how to allocate native types on the native heap and how to work with direct (unsafe) pointers compared to tracking handles (object references).

Marshaling (Interop)

- Primitive types are “naturally” marshaled
- Strings are the main problem

```
1 System::String^ s = "Hello World!";
2 pin_ptr<const wchar_t> p =
3     PtrToStringChars(s);
4
5 wchar_t* unicode = p;
6 char* ansi = (char*)(void*)
7     Marshal::StringToHGlobalAnsi(s);
8
9 System::String^ s2 = gcnew System::String(ansi);
10 System::String^ s3 = gcnew System::String(unicode);
```



A big question remains – how can complex types be marshaled across the managed-native boundary in C++/CLI? It’s fairly apparent that a C++/CLI project can interact with managed code as freely as it can interact with native code, but the marshaling problem remains a mystery so far.

Evidently, there is no effort involved in marshaling primitive types. As we saw before, converting a managed Int32 to the corresponding C++ int does not involve any explicit work from the programmer. However, non-primitive types such as strings often pose a problem, especially considering the immutability of managed strings and the ANSI-Unicode barrier. Lines 1-3 demonstrate how a managed string can be temporarily viewed as a collection of wchar_t, in other words – a **const wchar_t***. Note the constness – strings are immutable, and therefore there is no safe way to obtain a writeable pointer into the bowels of a managed string. Lines 5-7 demonstrate how a native wchar_t* can be converted to an ANSI string using the System::Runtime::InteropServices::Marshal class, namely the Marshal::StringToHGlobalAnsi method call. Remember that memory allocated and marshaled using the Marshal class should usually be freed by the Marshal::FreeHGlobal method.

Finally, lines 9-10 demonstrate that the managed string constructor can take a native ANSI string or a native Unicode string and construct a managed string representation on top of that (performing a copy of the original string to ensure immutability, of course).

Marshaling Framework

- `marshal_context`
- `TTo marshal_as(TFrom)`

If marshaling types by hand does not appeal to you, the marshaling framework distributed with C++/CLI in Visual Studio 2008 will seem like a much more powerful tool. It features the `marshal_context` class, which can be declared locally and managed the intricate details of allocating and freeing the actual representations of temporarily marshaled objects. Its `marshal_as` method (which is also available as a global function) has a number of template specializations which take a type and marshal it to another type. For example, a classic `marshal_as` specialization would take a `System::String^` and return a `const wchar_t*`.

CLR Types in the Native World

```
1 #include <msclr\marshal.h>
2 #include <msclr\marshal_cppstd.h>
3 using namespace msclr::interop;
4
5 class XmlInitializable {
6 private:
7     gcroot< XmlDocument^> _document;
8 public:
9     void Load(const std::string& fileName) {
10         marshal_context context;
11         XmlTextReader^ reader = gcnew XmlDocument(
12             context.marshal_as<String^>(fileName));
13         _document = gcnew XmlDocument();
14         _document->Load(reader);
15     }
16 };
```



This slide demonstrates how CLR (managed) types can be used in the native world. The `XmlInitializable` class is a native C++ class that is compiled to IL, but is not intended for use directly from managed languages. On the contrary – native C++ code that is *not* compiled to IL is free to use the `XmlInitializable` class, because it can be created without involving the GC heap. The `XmlInitializable` class, however, requires the use of the `System::Xml:: XmlDocument` class, which is a standard reference type from the .NET framework. Due to technical limitations, it cannot be embedded as a member of the `XmlInitializable` class directly, but by using the `gcroot` template (line 7), the `XmlInitializable` class can aggregate it and use it freely. The `Load` method depicted in lines 9-15 takes advantage of the `marshal_context` class and `marshal_as` method shown in the previous slide to marshal the file name, provided as a constant `std::string` reference to a `System::String^` which can be consumed directly by the `XmlTextReader` class.

Native Types In The CLR

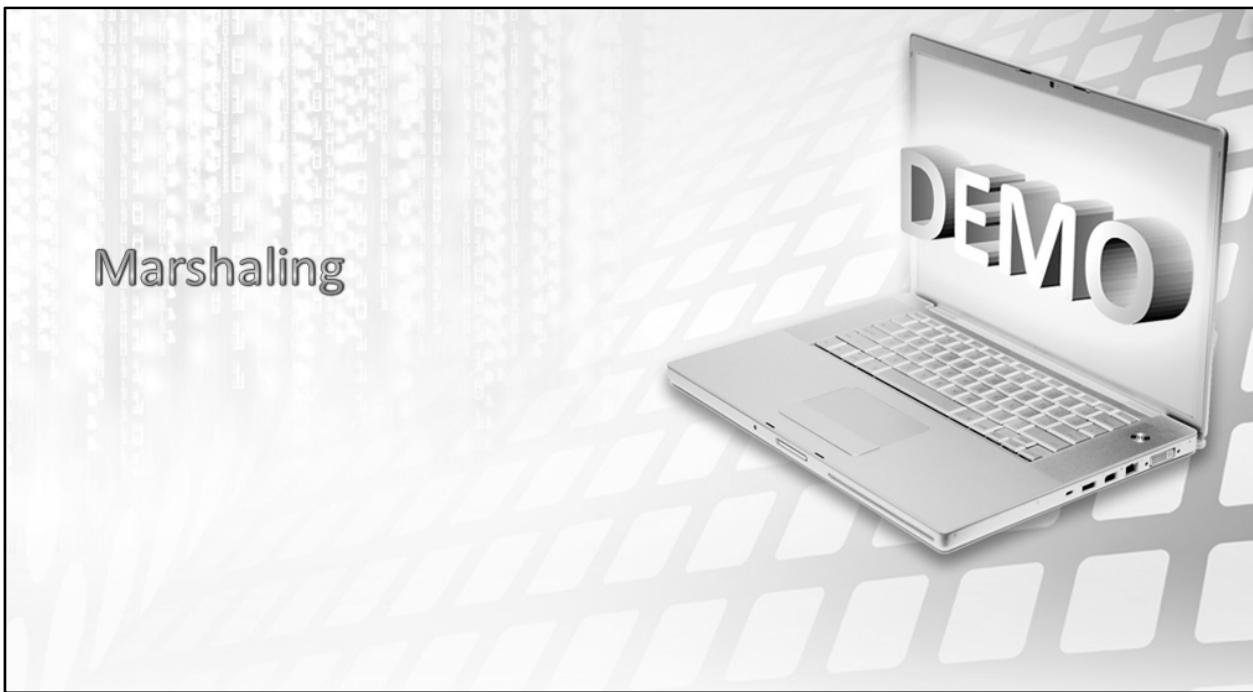
```
1 Permutations(IEnumerable<String^>^ strings) {
2     _strings = new vector<string>;
3     for each (String^ s in strings)
4         _strings->push_back(
5             _context.marshal_as<string>(s));
6 array<String^>^ Next() {
7     _hasNextPermutation = next_permutation(
8         _strings->begin(), _strings->end());
9     array<String^>^ list =
10        gcnew array<String^>(_strings->size());
11    for (it = _strings->begin();
12        it != _strings->end(); ++it) {
13        list[it - _strings->begin()] =
14            _context.marshal_as<String^>(*it);
15    }
16    return list; }
```

CODE

This slide demonstrates how native types can be used by managed types. In this example, the motivation for using native types in CLR code is the *next_permutation* STL algorithm which does not have a direct equivalent in the managed world. The *next_permutation* algorithm takes an input and produces the next permutation of the input in lexicographical order, until all permutations are exhausted.

The Permutations constructor and the Next method demonstrate the use of a vector of strings to contain marshaled copies of the source enumerable and the mechanism required to extract from the vector its contents and store them in an array which is then returned to the client.

Note that the class' client does not have to know that it is implemented using the *next_permutation* STL algorithm – as long as the class does not expose any C++/CLI specifics, it can be consumed from any managed language. In this case, the class interface is modeled around `IEnumerable<string>` and `string[]`, which are universal and can be consumed from any managed language.



See the **UsingCppCli** project in the **Module07_Interop** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how marshaling allows CLR types to interact with native types, and how CLR types can live in the native world and vice versa.

Uniform Destruction/Finalization

- Every type can have a destructor, `~T()`
- Every type can have a finalizer, `!T()`

The word "CODE" in a large, bold, white font with a black drop shadow.

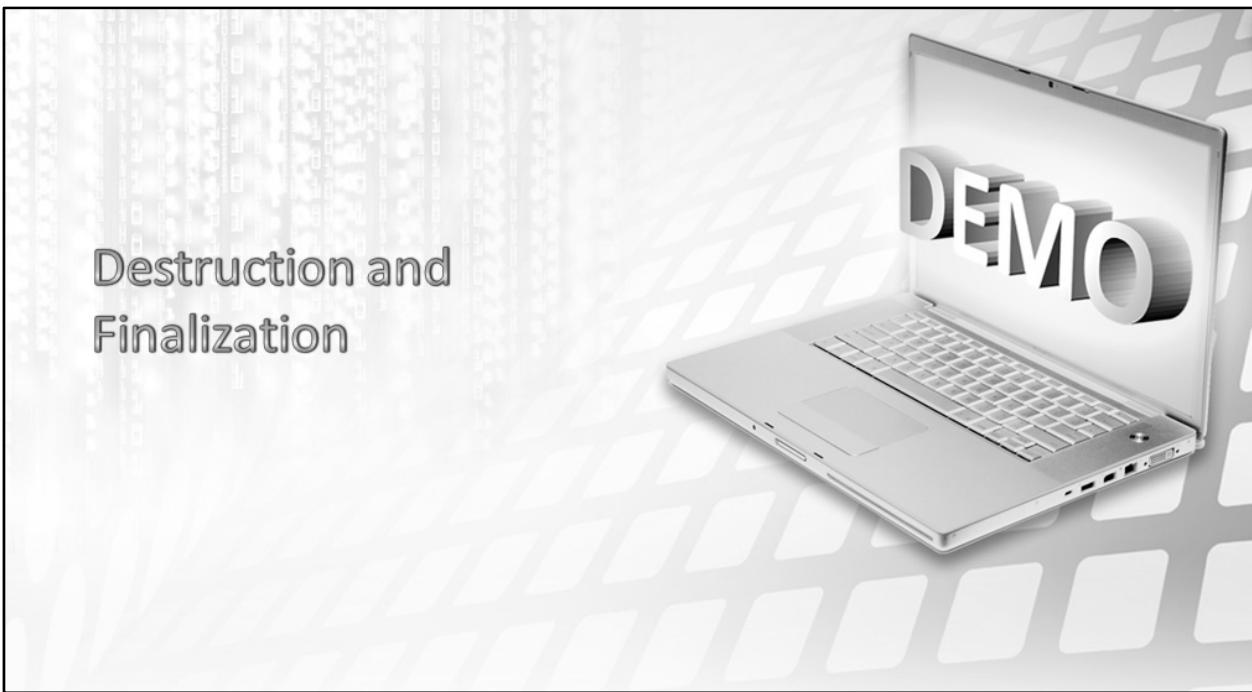
```
1 Permutations^ p1 = gcnew Permutations(EmptyArray);
2 delete p1;      //Calls the "destructor"
3
4 Permutations^ p2 = gcnew Permutations(EmptyArray);
5 //No explicit delete, so finalizer will be called later
6
7 {
8     Permutations p3(EmptyArray);
9     p3.HasNextPermutation;    //Direct access, no ->
10 }    //"destructor" called at this line
```

Finally (pun intended), it's time to discuss finalization and destruction of C++/CLI managed types. Similarly to C#, C++/CLI types can declare a finalizer, but the syntax is not similar – the `!` symbol is used to declare a finalizer. On the other hand, the `~` symbol (used to declare a finalizer in C#) is used to declare a *destructor*, which is nothing more than an implementation of the Dispose pattern. (See the GC module for more details.)

The non-trivial destructor denoted by the `~` symbol is implicitly run when a stack based object goes out of scope (as in line 8 and line 10). It is also run when the `delete` operator is invoked on a tracking handle (as in line 2). If neither occurs, the object will be finalized non-deterministically, which is the fate of the `Permutations` instance created in line 4.

As always, applications should strive to use deterministic cleanup and not rely on finalization. Finalizers should be used to assert failures in deterministic cleanup. (For more information, see the GC module.)

MCT USE ONLY. STUDENT USE PROHIBITED

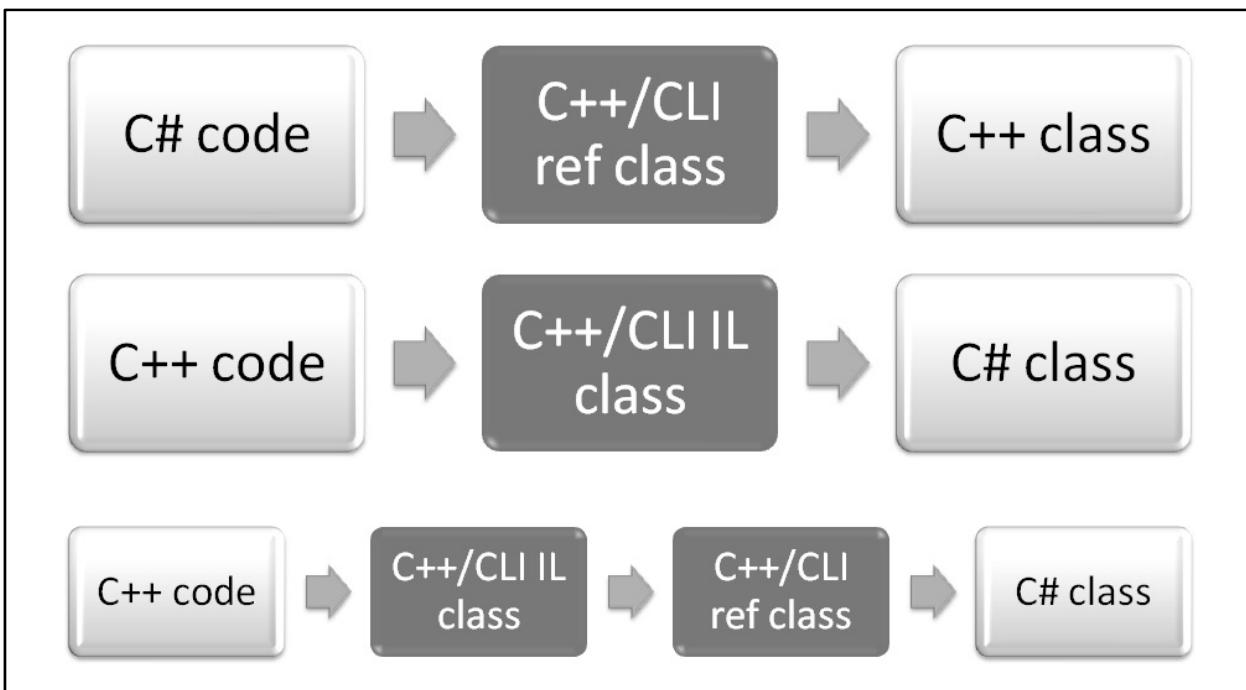


Destruction and Finalization

See the **UsingCppCli** project in the **Module07_Interop** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how C++/CLI managed types can declare finalizers and destructors to implement deterministic and non-deterministic finalization.

Practical Interop Scenarios



In this diagram, several practical scenarios for developing interoperability solutions are demonstrated. These solutions are unique to C++/CLI in that they would typically require significantly more work if implemented using COM Interop or P/Invoke. The primary reason for this is that C++/CLI makes it very easy to interact with C++ classes, COM objects and global “flat” methods – while the other interoperability technologies focus on only one of the aforementioned aspects.

- Wrap a native type with a C# class
 - In the middle: C++/CLI ref class
- Access a C# class from native code:
 - In the middle: C++ (not ref) class compiled with #pragma managed
- Events and callbacks may require two hops:
 - C++ native → C++ IL → C++ ref → C#
 - The reason for this is the inherent limitation of the event mechanism. Only a managed class (not simply a native class compiled to IL) can register to a managed event, thereby requiring another layer in between.

Lab: Native FileSystemWatcher Low-Fragmentation Heap Wrapper



Exercise 3 – C++/CLI: Native To Managed:

Wrap the `System.IO.FileSystemWatcher` class so that it can be used from native code. Expose the following native interface:

```
typedef void (*FS_NOTIFICATION_CALLBACK)(const std::string& path, const std::string& change);  
class NativeFileSystemWatcher  
{  
public:  
    NativeFileSystemWatcher(const std::string& path);  
    ~NativeFileSystemWatcher();  
    void AddCallback(FS_NOTIFICATION_CALLBACK callback);  
    void RemoveCallback(FS_NOTIFICATION_CALLBACK callback);  
};
```

Note: you cannot register a non-static member function of a native class to a managed event directly. You need to use an intermediate managed wrapper (consult the instructor).

You can find the solution for this lab in the **InteropExerciseSolution** solution under the **ExerciseSolutions\Module07_Interop** folder (specifically the **FileChangeWrapper** project).

Exercise 4 – C++/CLI: Managed To Native

Wrap the low fragmentation heap (LFH) Win32 API so that it can be used from managed code. Expose at least the following managed interface to provide a byte-array alternative that is allocated from the low-fragmentation heap:

```
public ref class LFHBuffer
{
public:
    static void InitializeLFH();
    LFHBuffer(int size);
    property System::Byte default[int]
    {
        System::Byte get(int index);
        void set(int index, System::Byte value);
    }
    property int Length
    {
        int get();
    }
    static LFHBuffer^ FromArray(array<System::Byte>^ buffer, int offset, int length);
    static LFHBuffer^ FromStream(System::IO::Stream^ stream, int length);
    void Read(array<System::Byte>^ buffer, int bufferOffset, int myOffset, int length);
    void Write(array<System::Byte>^ buffer, int bufferOffset, int myOffset, int length);
    ~LFHBuffer();
    !LFHBuffer();
};

};
```

You are welcome to extend the interface to provide efficient memory copy, memory search and other facilities. You can also implement an **LFHStream** class that extends **System.IO.Stream** and provides streaming capabilities based on an **LFHBuffer** instance.

Consult the MSDN for **HeapCreate** and **HeapSetInformation** to create a low fragmentation heap (available on NT 5.1 and higher only).

You can find the solution for this lab in the **InteropExerciseSolution** solution under the **ExerciseSolutions\Module07_Interop** folder (specifically the **LowFragmentationHeap** project).

C++/CLI Summary

- The most powerful .NET language
- Interop is easiest: It Just Works
- Absolute control over marshaling



Summarizing C++/CLI in only a couple of hours and a single exercise is not easy. This was just a sample of what C++/CLI can do for you, and there is much more to learn. Bear in mind that you can always use P/Invoke or COM Interop from within C++/CLI, so in fact it is a complete superset of all other interoperability technologies.

Other than being the most powerful .NET language, C++/CLI offers fine-grained control over marshaling and provides IJW interop.

Interoperability Considerations

- P/Invoke: C-style exported DLL functions
 - Partial marshaling customization
 - Good performance
- COM Interop: COM objects
 - Hardly any marshaling customization
 - Mediocre performance
- C++/CLI: Anything C++
 - Absolute control over marshaling
 - Best performance if you know what you're doing

Choosing the appropriate interoperability flavor is often not easy, especially if you have mastered all the various interoperability techniques. This is an attempt to provide general guidelines as to when to use each technology.

Choose P/Invoke when you need to call simple, global, “flat” C-style exported functions in a DLL. If you need only partial control over marshaling and adequate performance, P/Invoke is for you. Choose COM Interop if you’re interacting with COM objects or are willing to expose a managed class to COM clients. There is hardly any control over marshaling unless you’re willing to use ILDASM/ILASM and customize the interop assembly, or write the entire interop assembly by hand, inline in your code. COM transition performance is mediocre, and so will be the cost of your interop.

Finally, choose C++/CLI if you know what you’re doing. Exercising absolute control over marshaling allows you to perform multiple micro-optimizations to squeeze every tiny bit of performance out of the system. However, as always with C++, it’s also dauntingly easy to shoot yourself in the foot at the same time.

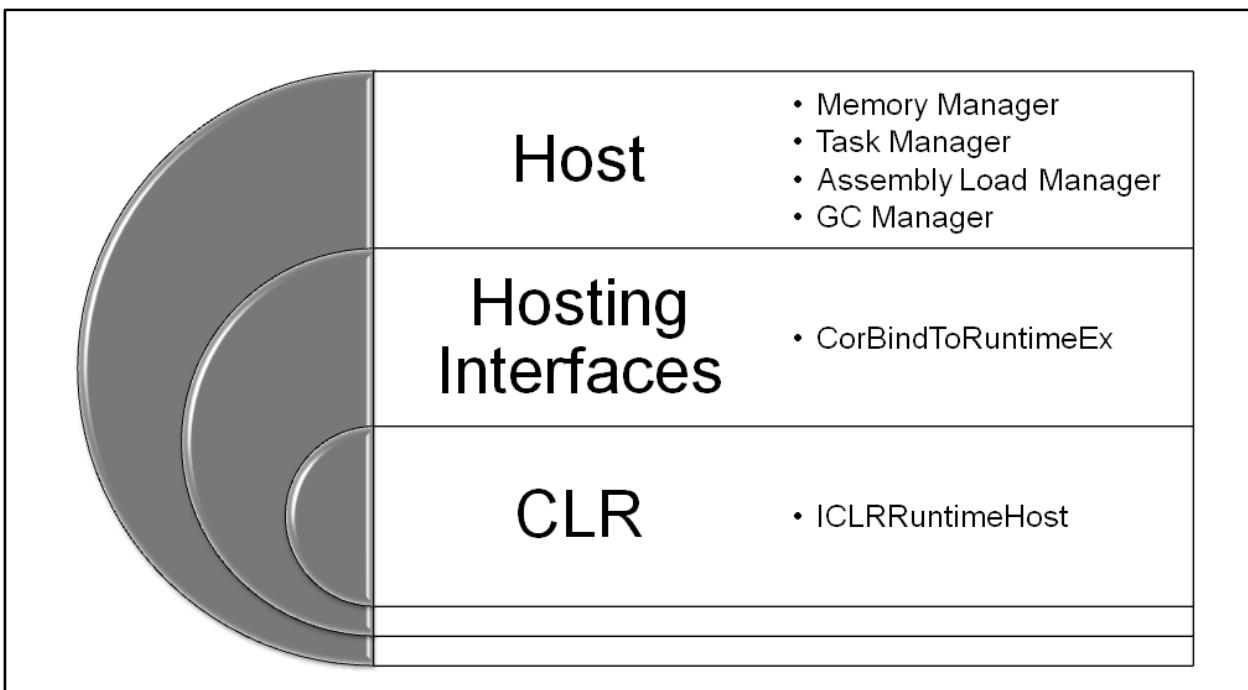
CLR Hosting

- Extremely powerful customization technique
- Host the CLR and tell it what to do
- The CLR relies on your services

Any discussion of CLR interoperability would not be complete without at least mentioning CLR hosting. CLR hosting is an extremely powerful customization technique which allows an unmanaged application to exercise complete control over CLR intrinsic behavior. Various services provided by the CLR to managed applications will be customized by your host, because the CLR contacts the host to perform these operations.

Using hosting interfaces, you can exercise fine-grained control over synchronization, task scheduling, memory allocation, garbage collection, assembly loading and various other features that cannot be customized from managed applications.

CLR Hosting From 10,000 ft



This course does not entail a discussion of how CLR hosting is implemented and does not show any example of CLR hosting. However, it is an extremely important and powerful technique to be aware of.

At a glance, an unmanaged application uses the `CorBindToRuntimeEx` API exported from `mscoree` to bind to the CLR and customize various hosting interfaces through the `ICLRRuntimeHost`. In turn, the CLR turns to the host when it requires a service such as memory allocation, assembly loading, implementing synchronization and various other tasks. Communication proceeds through well-defined channels in the shape of COM-style interfaces.

Summary

- Platform Invoke
- COM Interop
- C++/CLI
- Overview of CLR Hosting



In this module, we have reviewed three fundamental interoperability technologies which allow native and managed code to coexist and interact in the same application or process. We have also reviewed the pros and cons of the various technologies, from the ease of use, correctness and performance perspectives.



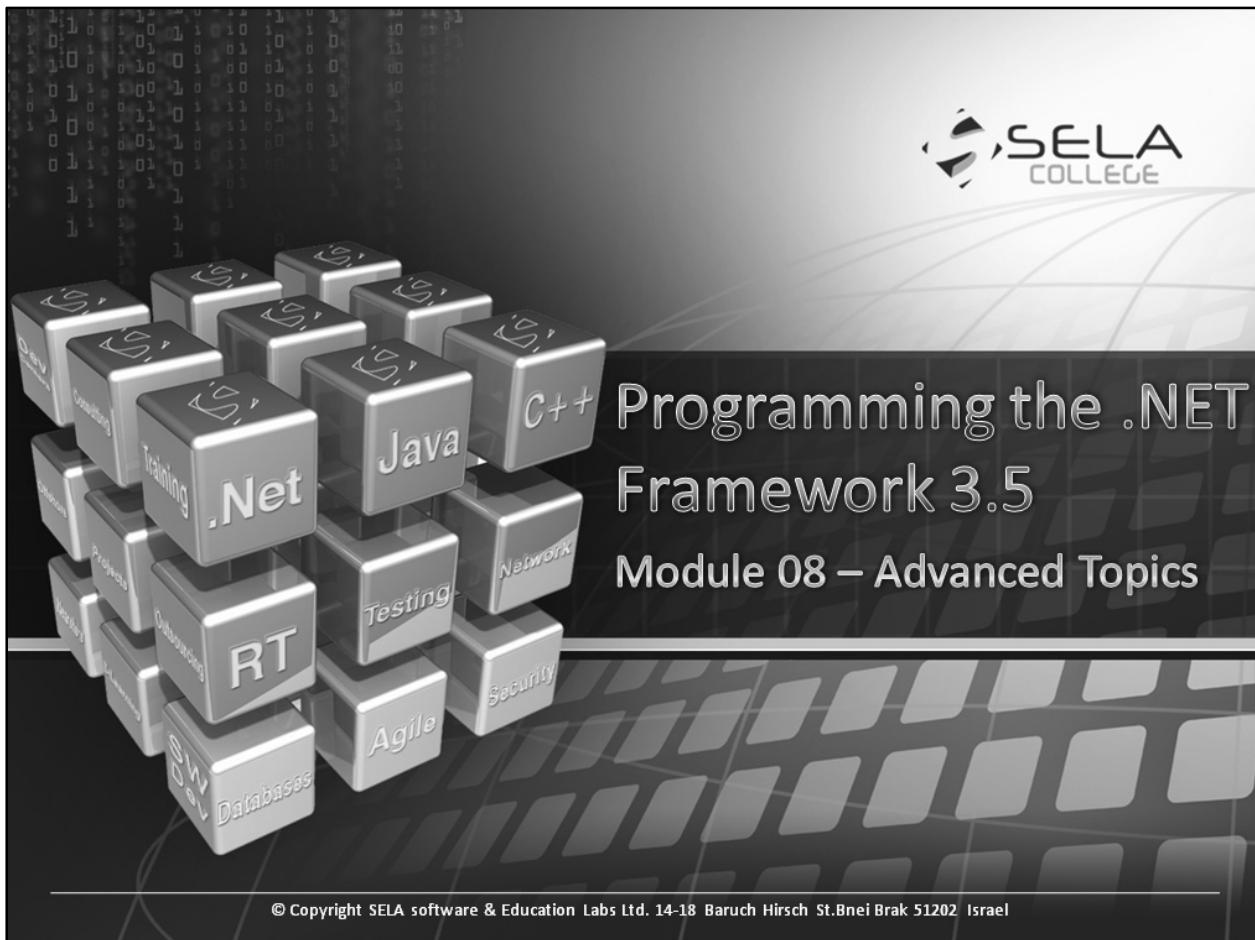
MCT USE ONLY. STUDENT USE PROHIBITED

Module 08 - Advanced Topics

Contents:

In This Chapter.....	3
.NET Startup Performance.....	4
Native Image Generator.....	5
NGEN Example.....	6
Dynamically Binding to Delegates	7
Dynamically Binding to Events.....	9
Event Registration Tricks	11
Invoking Events Asynchronously	12
Generics and Reflection.....	14
Generics at Runtime	16
Object Cloning as Serialization.....	18
Assembly Loading Diagnostics.....	19
Assembly Load Contexts	20
Troubles Faced with Load Contexts	21
.NET 3.5 SP1 Detailed Error.....	22
Summary	24

8-2 | Module 08 - Advanced Topics



In This Chapter

- ⌚ Improving startup performance with NGEN
- ⌚ Advanced delegates and events
- ⌚ Advanced generics
- ⌚ Object cloning as serialization
- ⌚ Assembly loading problems and contexts



This module is a collection of unrelated topics intended to expand your knowledge about various areas of the .NET framework. While not worthy of a module on their own, the topics discussed in this module might be very important in your day-to-day .NET development experience.

.NET Startup Performance

- What are the dominant factors in startup latency?
- Cold startup – I/O
- Warm startup – JIT compilation

User interface applications, especially those which start running along with the user's logging in, are especially susceptible to long startup latencies. An application that takes 5 seconds from double-click to main screen is something users are starting to mistrust, and as computing speeds grow the users' are becoming less and less patient.

The primary factors that dominate startup performance for managed applications are disk I/O and JIT compilation, two necessary ingredients on the startup path. While some applications might have other factors (which might diminish these two) such as network access during the startup path, most applications perform only I/O and JIT compilation when starting up.

The I/O costs of startup are mostly prevalent in the *cold startup* scenario, where the application is started for the first time since the system has booted up. However, in subsequent startup scenarios (*warm startup*) the I/O cost is greatly diminished because most of the I/O has already been cached by the system. This is especially true for loading the application code (assemblies), some of which are likely to remain cached because there might be other applications using them. (This is true for GAC assemblies in general and the assemblies comprising the .NET framework in particular.)

The warm startup performance, therefore, is dominated by JIT compilation, because every method accessed on the startup path must be compiled from IL to native code. This is an I/O- and CPU-bound process, which might take several seconds for a large application. Because JIT-compiled code is considered private to the process, it is thrown away when the application terminates and there is no way to reuse or cache it for subsequent invocations.

Native Image Generator

- NGEN pre-compiles IL to native code
- No JIT at runtime
- IL images still required (metadata)
- Automatically managed by NGEN service

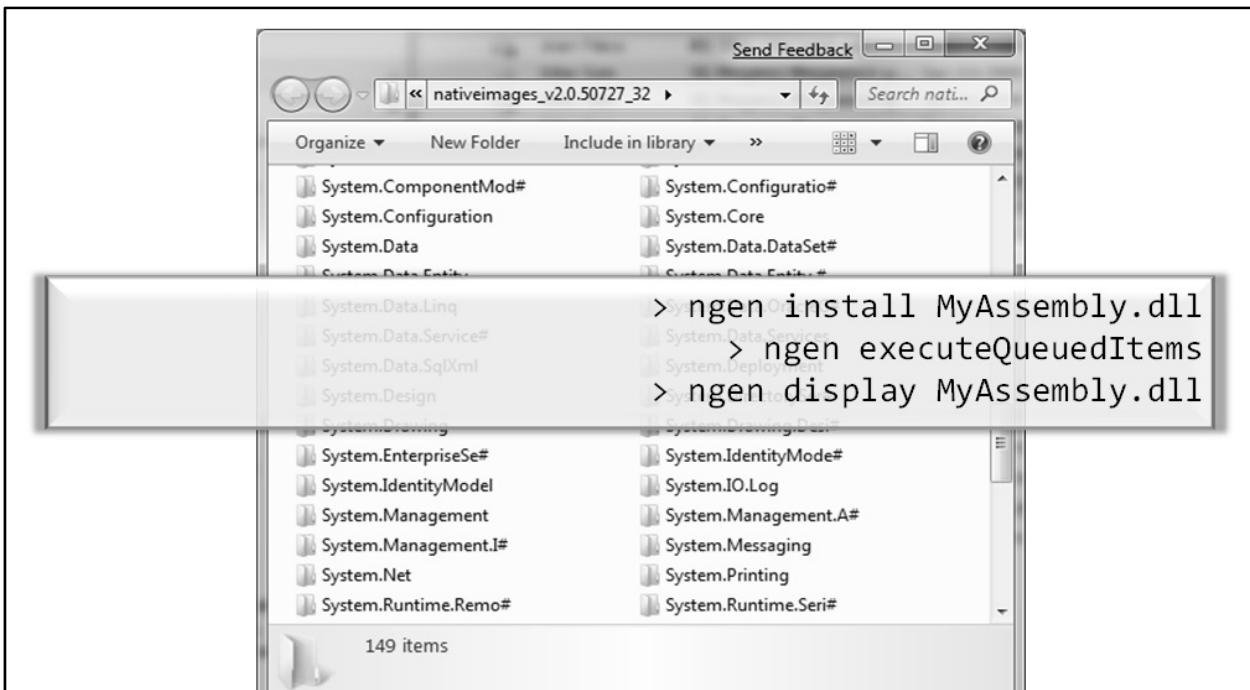
To address the problem of JIT-compilation costs in the warm startup scenario, the .NET framework is distributed with a Native Image Generator (NGEN for short, *ngen.exe*) which is capable of performing compilation of IL to native code before runtime, i.e. not just-in-time like the JIT does. If all application binaries were precompiled using NGEN, then there is no need to even load the JIT compiler at runtime and of course there is no need to compile any code on the startup path. This has a potential of significantly decreasing warm startup costs.

Running the native image generator on your application's assemblies does not imply that you can do away with the .NET framework installation. The original IL image must still reside on the target machine, because it is used for metadata and other purposes. Moreover, the native image generation can only be performed on the target machine – you cannot precompile the application in a development lab and distribute the precompiled results to customers. The process of generating, storing and managing the native images is well-managed by the .NET framework and the native image generation service, and is not subject to xcopy-deployment.

If you're considering to use NGEN in your application, remember this: It's highly unlikely that you will harm the performance of your application by using NGEN, but it's also impossible to guarantee that you will see any improvement in performance (especially for applications which do not have a startup latency problem). Nonetheless, all BCL assemblies are NGEN-d in advance when the .NET framework is installed on a particular machine, hinting that NGEN is a fairly common and useful approach.

8-6 | Module 08 - Advanced Topics

NGEN Example



This slide shows the demo command line commands necessary to generate a native image for an assembly called MyAssembly.dll. The ngen.exe tool is part of the .NET framework, and can be found under %windir%\Microsoft.NET\Framework (or %windir%\Microsoft.NET\Framework64 on a 64-bit machine, for the 64-bit version) within the latest CLR's folder.

The native images generated by the NGEN service are stored under %windir%\Assembly, where the GAC resides. These folders are not usually browsable in Windows Explorer, and there is no particular reason to inspect them manually. This screenshot shows some of the folders under the NativeImages folder for expository purposes.

Dynamically Binding to Delegates

- ① Delegate target unknown during compilation
- ② `Delegate.CreateDelegate`

```
1 logGenerator =  
2     (LogMethodDelegate)Delegate.CreateDelegate(  
3         typeof(LogMethodDelegate), @object, logMethod);
```



Our next topic is dynamic binding and creation of delegates. In the C# courses, you are usually taught to bind and create delegates statically (at compile-time) by specifying their signature and their target method. However, oftentimes the ability to dynamically create a delegate and bind it to a method (e.g. using Reflection) might be greatly appreciated.

This can be done using the `Delegate.CreateDelegate` method, which can bind any delegate type to a static or instance method. One of its overloads takes a delegate type, an object instance and a `MethodInfo` object (that can be obtained using Reflection) and returns a delegate of the specified type which references the appropriate method on the specified object. The resulting delegate, after casting down to the specific delegate type, can be invoked directly without the overhead of reflection or dynamic invocation. It is only slightly more expensive than a direct method call, and is comparable with an interface call.

MCT USE ONLY. STUDENT USE PROHIBITED



Dynamic Delegates and [LogMethod]

See the **DelegatesAndEvents** project in the **Module08_AdvancedTopics** solution under the SampleCode folder for more information about this demo in the code comments. (Specifically look under the LoggingFramework project folder.)

In this demo, you will see how dynamic binding to delegates at runtime allows for interesting features that cannot be statically accomplished. The demo shows a logging framework which is based on a “log method” responsible to output the object’s log-friendly representation at runtime. The log method is decorated with a `[LogMethod]` attribute, and when the logger (implemented in `DynamicLogger.cs`) is invoked, it looks for the method decorated with this attribute and binds a delegate to it dynamically. The binding improves performance significantly because the result is cached and future invocations use the delegate directly without the Reflection overhead.

Dynamically Binding to Events

- ① Event source or handler unknown during compilation
- ② `EventInfo.AddEventHandler`

```
1 Delegate handler =
2     Delegate.CreateDelegate(
3         @event.EventHandlerType, @object, method);
4 @event.AddEventHandler(null, handler);
5 //SystemEvents is a static event source
```



Similar to runtime binding of delegates, events can be bound to at runtime as well. We already know how to bind a delegate to a method at runtime (which is necessary to produce the event handler for the event), but binding to an event that is not statically known is a different matter. Using Reflection, we can obtain the `EventInfo` instance which describes the event, and then use the `AddEventHandler` and `RemoveEventHandler` methods to register a new handler to the event. The handler itself, as a twist, can be created dynamically as before. Note that the first parameter of `AddEventHandler` is not the object that is registering to the event – it is the event source (because an `EventInfo` instance can be created from a type without even having an instance of it handy). For static events, such as in the example on the slide, there is no need to provide an object instance for this parameter.



Dynamic Events and [RegisterSystemEvent]

See the **DelegatesAndEvents** project in the **Module08_AdvancedTopics** solution under the SampleCode folder for more information about this demo in the code comments. (Specifically look under the SystemEventsFramework project folder.)

In this demo, you will see how dynamic registration to events enables an interesting scenario where an object indicates its interest in a certain event and is automatically wired to that event by the underlying framework. Specifically, the [RegisterSystemEvent] attribute is placed on type methods which express interest in certain system events (from the Microsoft.Win32.SystemEvents class), such as changing the display settings or the end of the session. The underlying event framework creates the appropriate delegate for the method and adds the event handler to the appropriate event (by discovering the event itself dynamically, by its name, using Reflection).

Event Registration Tricks

- Register with an anonymous method
- Register with a lambda
- Register with delegate { }

Several more remarks about events might make your life easier programming them:

- You can register an anonymous method to an event just as easily as any other method.
- You can register a lambda expression to an event just as easily as any other method.
- When authoring events, you can set the event's initial value (at the field initialization statement) to "delegate {}", which assigns to the event one empty handler which does nothing. This allows you to later invoke the event without first checking if it's null, because it will never be null (because of the empty handler).

(A detailed explanation of anonymous methods and lambda expressions is outside the scope of this module. See the C# courses for more information.)

Invoking Events Asynchronously

- ① MulticastDelegate.GetInvocationList
- ② Invoke each handler asynchronously

```
1 public void Invoke(params object[] args)
2 {
3     foreach (Delegate del in _del.GetInvocationList())
4     {
5         ThreadPool.QueueUserWorkItem(
6             delegate { del.DynamicInvoke(args); });
7     }
8 }
```

CODE

In certain event registration scenarios, it's extremely important to be able to invoke the event handlers asynchronously. For example, if the event is a low-latency phenomenon such as a system timer that occurs several times each second, then it's impossible to tolerate the latency of the event handlers executing, risking to lose the time sync with event invocations.

Invoking an event (or any delegate) asynchronously is fairly easy, if you remember that each event and delegate derive from `MulticastDelegate`, providing a convenience method called `GetInvocationList` which returns an array of registered handlers. Each of these handlers can then be invoked individually, as shown in lines 5-6 in the slide. (By the way, if you have prior information about the delegate, you can cast the returned `Delegate` objects to the specific delegate type, and invoke them directly instead of using the `DynamicInvoke` method. This will improve performance significantly, even though in this scenario the likely dominating cost is the asynchronous invocation on the thread pool.)



See the **DelegatesAndEvents** project in the **Module08_AdvancedTopics** solution under the SampleCode folder for more information about this demo in the code comments. (Specifically look under the AsyncDelegates project folder.)

In this demo, you will see how delegates and events can be invoked asynchronously. The `AsyncInvoker` class represents a wrapper which asynchronously invokes delegates (similar to the code you've seen on the previous slide). The `AsyncEventSource` class represents an asynchronous event source, which publishes the event asynchronously to its subscribers. The example does not use the classic C# events mechanism, but the use of delegates and similarly named methods (`Add`, `Remove`, `Invoke` etc.) makes it easy to convert one construct to the other.

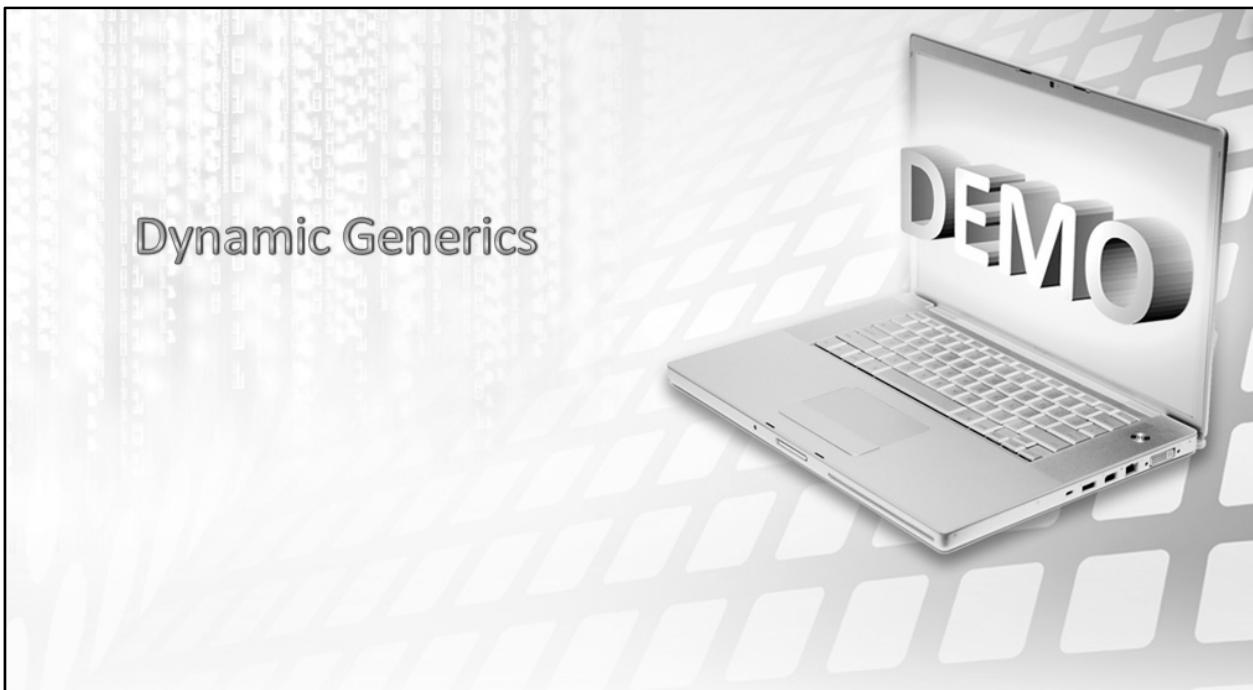
Generics and Reflection

- Open generic type, closed generic type
- `Type.MakeGenericType`,
`Type.GetGenericArguments` etc.
- Dynamically create generic types (WCF channel, ...)

Our next subject is generics. You have seen the fundamentals of generic syntax in the language introduction (C#) courses, so there are only a few minor advanced topics to discuss here. First is the important distinction between *open generic types* and *closed generic types*. An open generic type is the `List<T>` canonical example – it is not a type of which instances can be created – it is a “template” from which additional types can be cloned. A closed generic type is the `List<int>`, `List<float>`, `List<string>`, `List<object>` and the rest of the infinite set of types which are associated with the open generic type. In C#, the open generic type can be obtained using the odd `typeof(List<>)` syntax, omitting the generic parameter names. (A similar example for a Dictionary would be `typeof(Dictionary<,>)`).

Using Reflection, it’s possible to construct a closed generic type from an open generic type, and to retrieve the open generic type associated with the closed generic type. The first ability is more useful, and is possible thanks to the `Type.MakeGenericType` method which takes an open generic type, adds to it the provided type parameters and returns a closed generic type of which instances can be created.

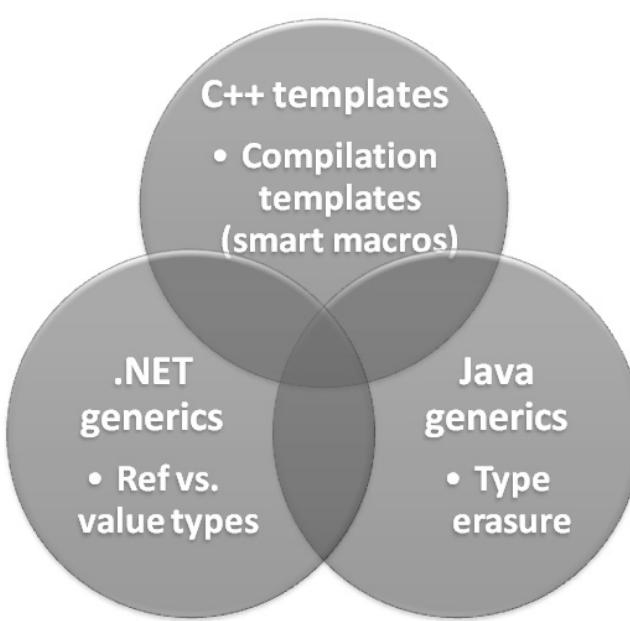
This is extremely useful in a variety of scenarios, particularly when an API is designed with generics in mind but you need to circumvent the static nature of generics and bind to the API at runtime, using a `Type` instance you discover dynamically. For example, when using the WCF APIs for creating a channel, you would need a `ChannelFactory<T>` which returns `T` from its `CreateChannel` call. Binding to the channel factory dynamically at runtime using only a `Type` instance is possible using the `Type.MakeGenericType` method described above.



See the **DynamicGenerics** project in the **Module08_AdvancedTopics** solution under the SampleCode folder for more information about this demo in the code comments.

In this demo, you will see how by using the Reflection generics-related APIs it's possible to bind a generic type and a generic delegate using type information available only at runtime. The demo reads type information from the console (the user), and obtains a Type instance dynamically, proceeding to create a generic type instance and bind to it a generic delegate using the same Type instance.

Generics at Runtime



A more detailed description of how generics work at runtime is in place. But first, we have to take a look at how C++ templates and Java generics (two related mechanisms) work at runtime, so that we can appreciate the pros and cons of the .NET implementation.

C++ templates are a generic programming construct that is applied only during compilation. After compilation, the C++ runtime does not know the relation between the generic `list<T>` type and its specific `list<int>` instantiation. There is no mention of the generic type in the compilation artifacts, and the linker is only aware of specific template instantiations. Templates in C++, therefore, are a form of smart macros, which are expanded on demand by the compiler to produce more instances of generic types and methods. This approach has the significant advantage of incurring absolutely no runtime overhead for individual types, but also has the significant disadvantages of code bloat (multiple instances of the same templated type in the same library) and of code sharing (it's impossible to export templated types across DLL boundaries).

Java generics are a generic programming construct that is completely different from C++ templates. A Java generic type will have its generic part erased during compilation, so that at runtime all generic instances behave as if they were using `java.lang.Object` wherever a generic parameter placeholder used to be. This implementation has the significant disadvantage that there is no performance benefit gained by using generic types, and that it's impossible to use generic types with primitive values – their wrappers must be used instead. (I.e. `Integer` instead of `int`, and so on, producing a boxing and unboxing cost and a bigger heap signature.)

Finally, .NET generics are a hybrid solution which takes advantage of metadata and runtime compilation. A .NET generic type does not have its type information erased, nor is the generic information replaced by the compiler. A .NET assembly (in IL) has the notion of the open generic type and the closed generic types that are associated with it, and the type object and

compilation information of the closed types is generated only on demand (just-in-time). On the other hand, all closed generic types which use reference type parameters have their code shared (because all references have the exact same size in native compiled code) – reducing the runtime footprint of generics. All closed generic types which use value type parameters have a separate version of the code, to allow for specific optimizations for different value types and to accommodate the fact that value types have different sizes and associated instruction sets.

Object Cloning as Serialization

- ① Shallow clone: `Object.MemberwiseClone`
- ① Deep clone: `ICloneable.Clone`
- ① Cloning as serialization

```
1 public static T Clone<T>(this T obj)
2 {
3     using (MemoryStream stream = new MemoryStream())
4     {
5         _formatter.Serialize(stream, obj);
6         stream.Position = 0;
7         return (T)_formatter.Deserialize(stream);
8     }
9 }
```

CODE

Our next brief detour goes into the world of serialization. In Module 04, `Serialization` we have seen many uses for serialization, but one use we have no mentioned is object cloning. There are various ways in .NET to clone an object, the most shallow and simple one being the protected `Object.MemberwiseClone` method that every type has. However, deep cloning is something left to the programmer's discretion, and is usually implemented by a series of exhausting assignments for every field of the object.

An alternative form of cloning is presented in the slide – it's a generic extension method called `Clone` which takes its parameter, serializes it into a memory stream and deserializes it back to produce an exact (deep) copy of the original instance. While this approach has a higher performance cost because of the serialization overhead, it might be a very appropriate solution for large objects which are rarely cloned but where a manual implementation would consume a significant amount of time and introduce boilerplate code.

Assembly Loading Diagnostics

- .NET assembly loading is not a straightforward process.
- It involves GAC, private probing, LoadFrom ...
- Use *fuslogvw* to the rescue

Our last subject in this module is .NET assembly loading and associated diagnostics. There are numerous ways to load a .NET assembly, and the general process of binding to assemblies is not a simple process at all. Factors such as whether a full or a partial assembly name was provided, whether the assembly has a strong name, whether the assembly is in the GAC or not, whether there is a private probing path configuration – and many others – affect the binding process. If you're getting confused over what is loading where and why, you should first read the theoretical introduction to assembly binding (which can be found on the MSDN) and then use the excellent utility for diagnosing assembly binding called *fuslogvw* and available from the Visual Studio command prompt. When properly configured, the tool will let you know of every assembly bind (successful or failed) that occurs on the machine, giving you the opportunity to diagnose and pinpoint assembly binding problems.

Assembly Load Contexts

- Two forms of assembly loading:
- “Standard” load
 - JIT, Assembly.Load, implicit reference
- “Load-from” load
 - Assembly.LoadFrom

One specific binding scenario that might be excruciatingly hard to debug and diagnose has to do with assembly load contexts. (A more detailed description of the feature and the problem scenario is available at <http://blogs.microsoft.co.il/blogs/sasha/archive/2007/03/06/Assembly-Load-Contexts-Subtleties.aspx>)

There are two primary forms of assembly binding in the .NET loader nomenclature: “Standard” or “default” loading and “load-from” loading. For each form of binding there is a separate *load context*, which is a repository of assemblies. The amazing thing is that the same assembly (with the same name and even the same physical location) *might* be loaded in *both* load contexts – the default load context and the load-from load context.

How does an assembly end up in one of the contexts?

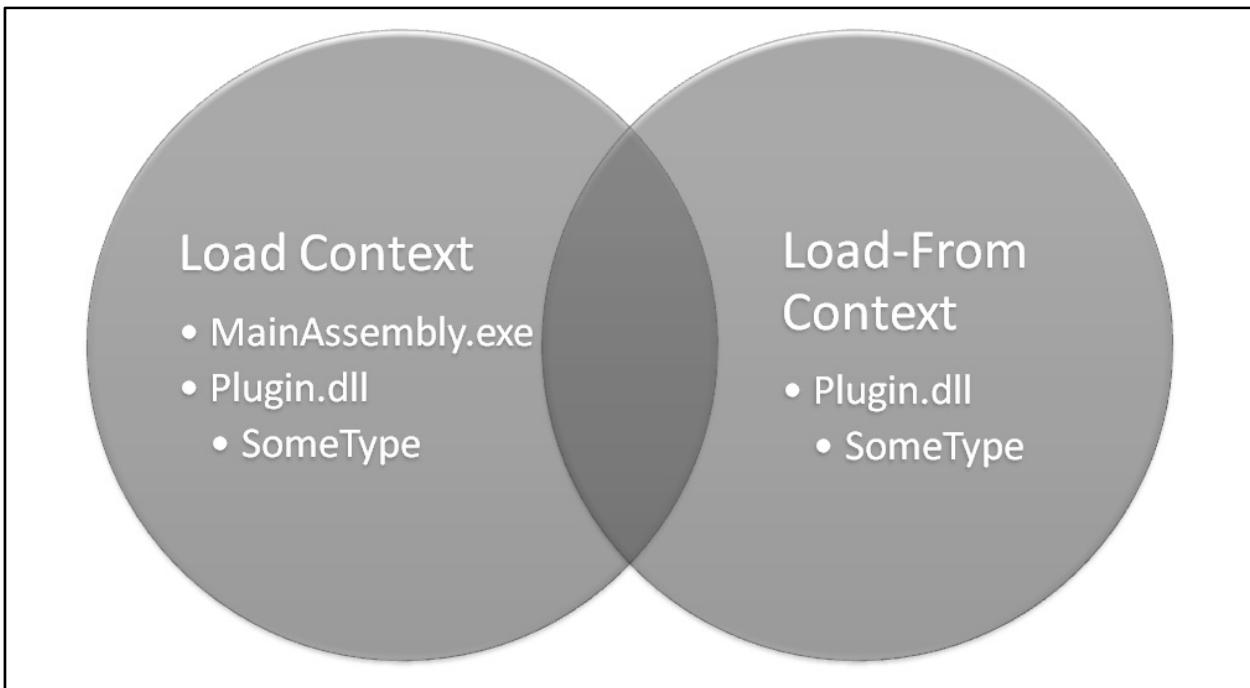
If the assembly was loaded in a standard way, such as being automatically resolved by the JIT, using the Assembly.Load method and other alternatives – then the standard binding process applies and the assembly ends up in the default load context.

However, if the assembly is loaded using the Assembly.LoadFrom API, which takes an assembly location instead of a name, then it *might* end up in the load-from context. (The precise description of when that would happen is outside the scope of this module – consult the blog post referenced above for more information.)

Why is it important if an assembly ends up in one context or another? Because types from the *same assembly* loaded in two different contexts are not considered compatible. The diagram on the following slide demonstrates this:

As a remark, there is another form of assembly binding called “reflection-only assembly load” (loading an assembly for inspection only within a reflection-only context) and it is outside the scope of this module.

Troubles Faced with Load Contexts



In this diagram, we have a load context with the MainAssembly.exe assembly and another assembly called Plugin.dll. In that assembly there is a type called SomeType. In the load-from context we have the same assembly Plugin.dll with the same SomeType type, but the two types are not considered compatible!

Attempting to cast from the “first” SomeType instance to the “second” SomeType instance would produce a runtime error, which might be very difficult to decipher. Before .NET 3.5, you would get an error message along the lines of “Cannot cast SomeType to SomeType”, misleading you completely. The .NET 3.5 error message is more detailed and gives you the information on the various load contexts involved and how the assemblies ended up in these load contexts.

.NET 3.5 SP1 Detailed Error

Unhandled Exception: System.InvalidCastException:

[A]Plugin.MyPlugin cannot be cast to [B]Plugin.MyPlugin.

Type A originates from 'Plugin, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null' in the context 'LoadFrom' at location '`..\Plugin.dll`'.

Type B originates from 'Plugin, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null' in the context 'Default' at location '`..\Plugin.dll`'.

CODE

This is the error message produced by the .NET 3.5 runtime when the above scenario occurs. It is a snipped exception message from the actual demo project you will see in the subsequent demonstration.

fuslogvw and Assembly Load Contexts



See the **MainHost** project in the **Module08_AdvancedTopics** solution under the **SampleCode** folder for more information about this demo in the code comments. (And the rest of the projects under the **AssemblyBinding** solution folder, including **Plugin** and **SharedInterface**.) In this demo, you will see how load contexts play an important part in dynamic assembly loading scenarios. Specifically, you will see how the same assembly ends up loaded into the default load context and the load-from context, producing a difficult to understand exception with regard to incompatible types from the two copies of the assembly.

Summary

- Improving startup performance with NGEN
- Advanced delegates and events
- Advanced generics
- Object cloning as serialization
- Assembly loading problems and contexts

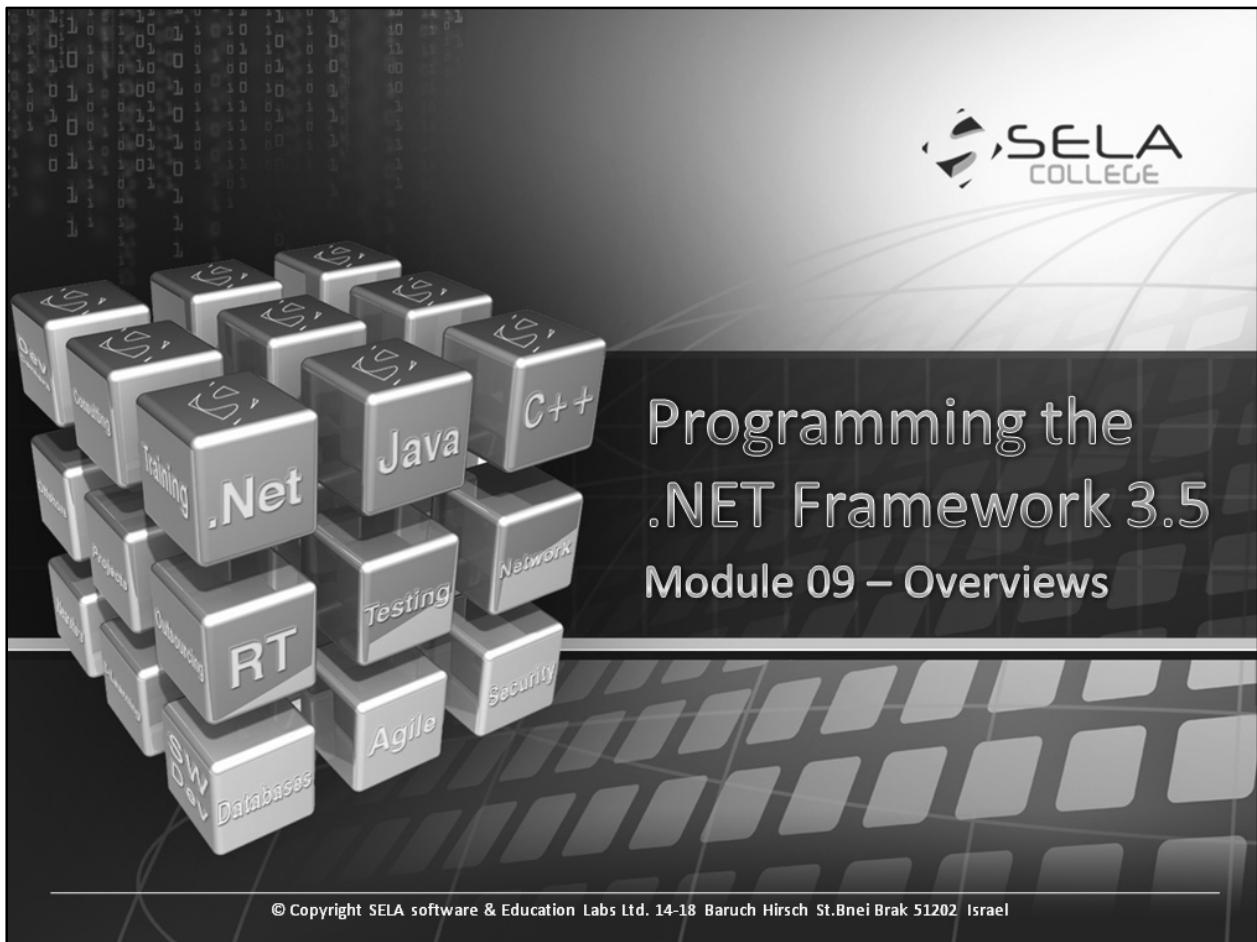




Module 09 - Overviews

Contents:

In This Chapter.....	3
ADO.NET	4
System.Transactions.....	5
Windows Communication Foundation	6
Windows Communication Foundation (contd.)	7
Windows Workflow Foundation.....	8
Language Integrated Query	10
Related Courses	11
Summary	12



In This Chapter

- ⌚ Overviews only, this is not a course
- ⌚ ADO.NET
- ⌚ System.Transactions
- ⌚ WCF
- ⌚ WF
- ⌚ LINQ
- ⌚ Related courses



This module provides an overview of various technologies that are related to or are part of the .NET Framework distribution. The purpose of this module is not to teach or even provide a feature tour of every technology, but merely to acknowledge their existing. Other Sela courses provide more information, training and practical labs on each subject mentioned in this module.

ADO.NET

- Accessing data using a set of classes
- RDBMS, XML



ADO.NET is a collection of classes that provides untyped and typed access to a relational database or a structured XML file. The DataSet class and the DataTable objects that it contains can be connected to the relational database using a DataAdapter. This produces a snapshot of the database state that can then be manipulated in memory.

ADO.NET also features a non-snapshot mode, in which a DataReader can be used to iterate the results of a query or even entire tables, record after record.

System.Transactions

- ① Managed abstraction of a transaction
- ① Isolation options, timeout options

```
1 //Access transactional resources in a TX scope
2 using (new TransactionScope())
3 {
4 }
```

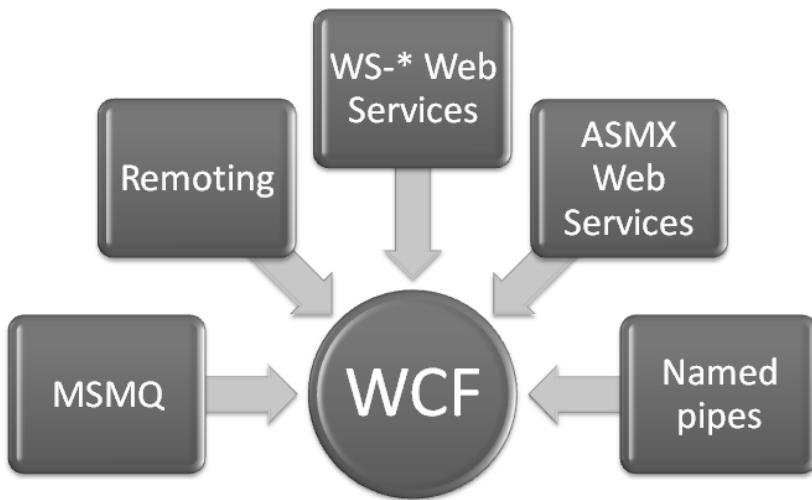
CODE

The System.Transactions assembly (which contains classes in the System.Transactions namespace) provide a managed abstraction of a local or distributed transaction. A transaction is a set of operations that have ACID properties (atomicity, consistency, isolation and durability) and provide great value to enterprise distributed applications. Accessing data within a transaction becomes a matter of creating a new TransactionScope object and calling the Commit method to commit the transaction. Additional options are provided by the Transaction.Current property, which retrieves the ambient local or distributed transaction even if invoked outside the system boundary. Data access and communication frameworks modeled on top of .NET 2.0 use System.Transactions. Among them you will find LINQ, WCF, WF and others.

MCT USE ONLY. STUDENT USE PROHIBITED

Windows Communication Foundation

- Unified framework for .NET communication

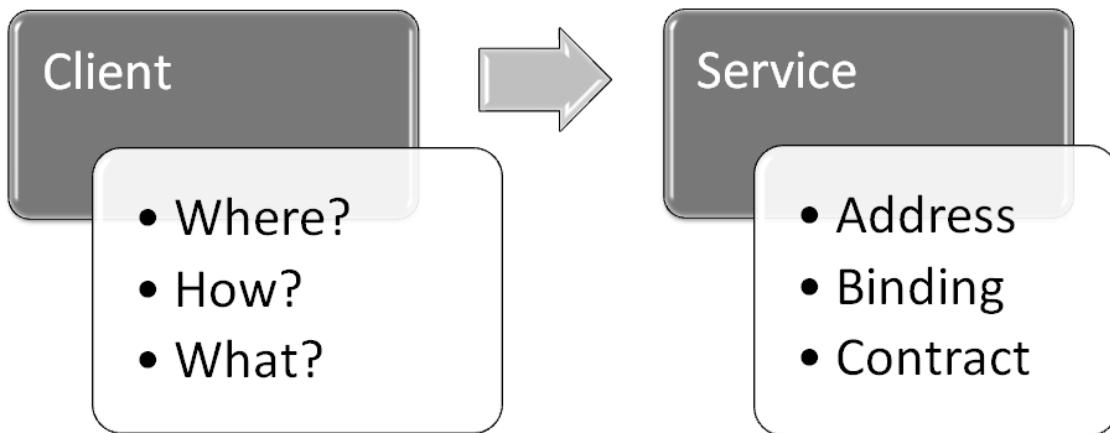


WCF (Windows Communication Foundation) strives to unify all communication frameworks that were part of .NET before the release of .NET 3.0. WCF features channels for binary work over TCP (resembling .NET Remoting), MSMQ, WS-* compliant web services, basic ASMX web services, named pipes and many other transports.

WCF also features a variety of extensibility points, providing developers with the opportunity to write a custom channel, a custom serializer, a custom formatter and to intercept events in the WCF message handling pipeline for logging, introspection, validation and many other purposes. WCF's excellent performance characteristics and widely acknowledged usability spectrum have gained it immense popularity since the release of .NET 3.0.

Windows Communication Foundation (contd.)

- Relies on SOA principles
- Not another Object-RPC

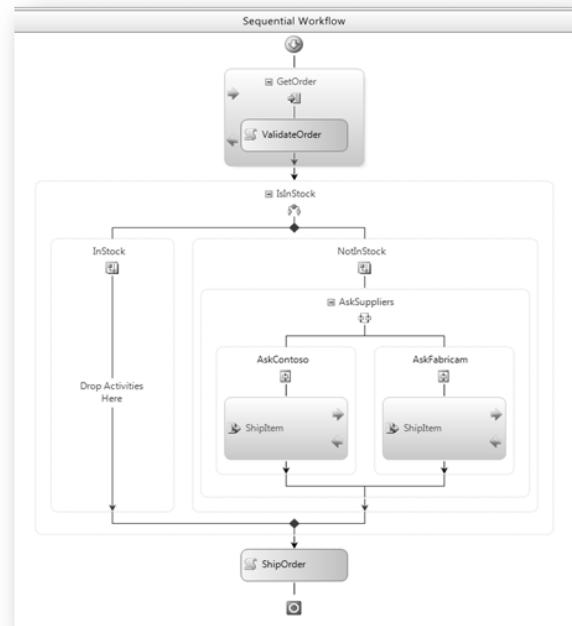


WCF communication reflects the SOA (Service Oriented Architecture) principles. As such, communication across WCF channels proceeds according to a well-defined schema. WCF is not an Object-RPC (Remote Procedure Call) technology – object references cannot be exchanged over WCF channels, only schema (data) can be exchanged.

A WCF endpoint is defined by ABC – an address which defines the physical location of the service, a binding which defines the shape of the communication with the service (the protocol, the encoding, the encryption and so on) and a contract which defines what data is exchanged and what operations the service supports. As long as the client and the service agree on a specific ABC, communication can proceed freely, even if one of the sides is not WCF-aware or even not .NET-aware.

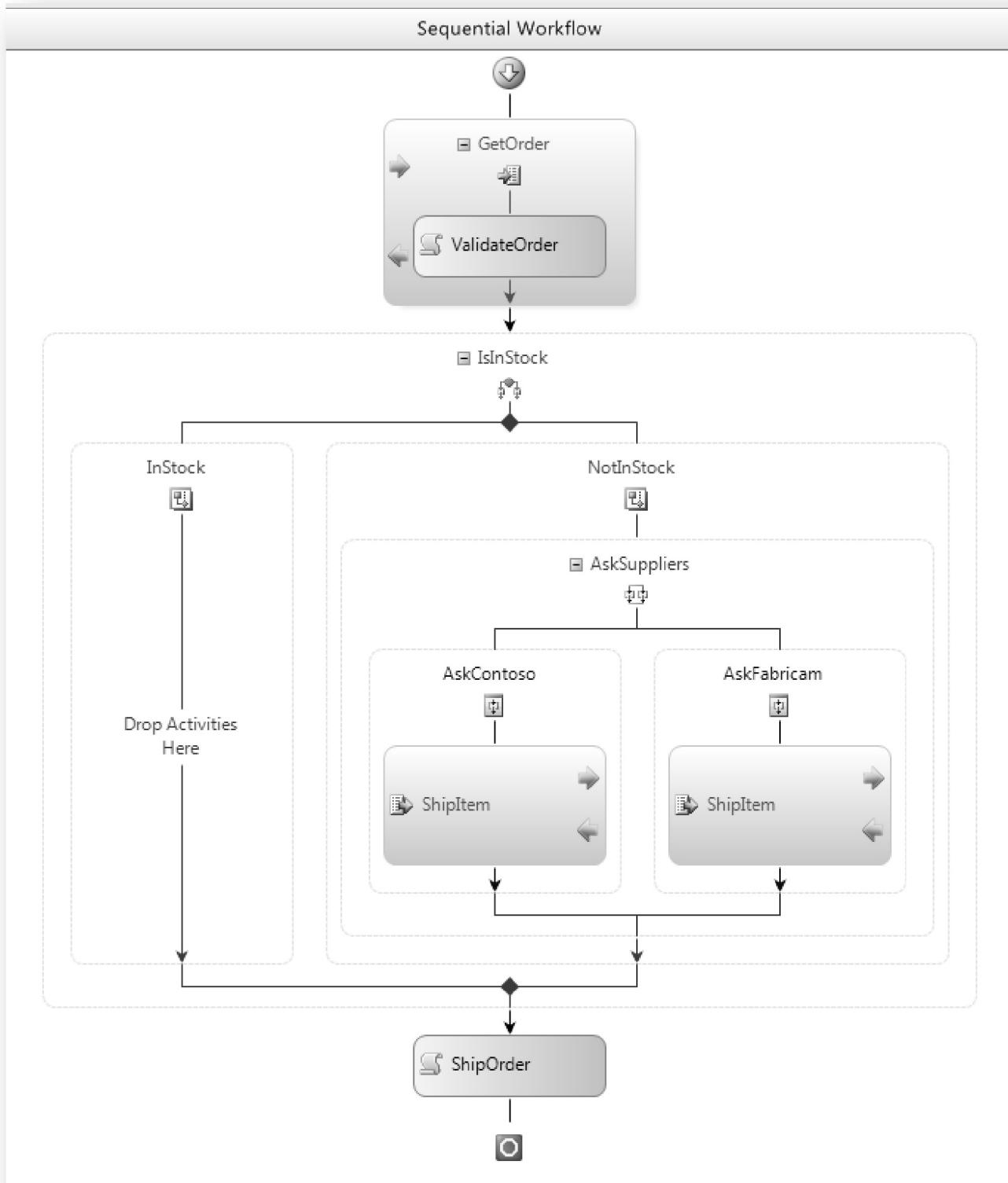
Windows Workflow Foundation

- Declarative modeling of Workflows
- Domain-specific building blocks (*activities*) form a *workflow*



WF (Windows Workflow Foundation) is a declarative framework for modeling workflows which consist of building blocks called activities. The workflow model includes hosting features for persistence of workflows, which allows for long-running workflows to span multiple physical machines and consume less resources when idle.

WF ships with .NET 3.0 and features a rich designer model which allows for building DSLs (Domain Specific Languages) expressed as activities as well as rules which define the shape of interactions between activities.



Language Integrated Query

- ① Data access framework which works with any data source
- ② LINQ to Objects, SQL, XML, Entities ...

```
1 var bulkCustomersByCity = from c in customers  
2                     where c.Orders.Length > 10  
3                     group c by c.City into g  
4                     select g;
```

CODE

LINQ (Language Integrated Query) is a framework for accessing data in a uniform, language-supported manner while abstracting away the data source. LINQ access to objects (collections), relational databases, XML files or any other data source depends on the availability of LINQ providers. The LINQ framework (part of .NET 3.5) ships with built-in providers for in-memory collections (LINQ to Objects), SQL Server databases (LINQ to SQL), XML documents (LINQ to XML) and others. Additional providers have been developed by third party vendors and by the development community.

A LINQ query is declarative by nature, in that it specifies the *how* of query execution and not the details of executing it.

Related Courses

- Windows Communication Foundation
- Windows Workflow Foundation
- LINQ via C# 3.0 (CWL #50066)

Other Sela solutions provide in-depth information, training, samples and hands-on labs on a variety of .NET Framework related subjects. Among them you will find courses on the Windows Communication Foundation, the Windows Workflow Foundation and the excellent LINQ via C# 3.0 which is also distributed through the Microsoft Courseware Library.

MCT USE ONLY. STUDENT USE PROHIBITED

Summary

- Overviews only, this is not a course
- ADO.NET
- System.Transactions
- WCF
- WF
- LINQ
- Related courses





MCT USE ONLY. STUDENT USE PROHIBITED