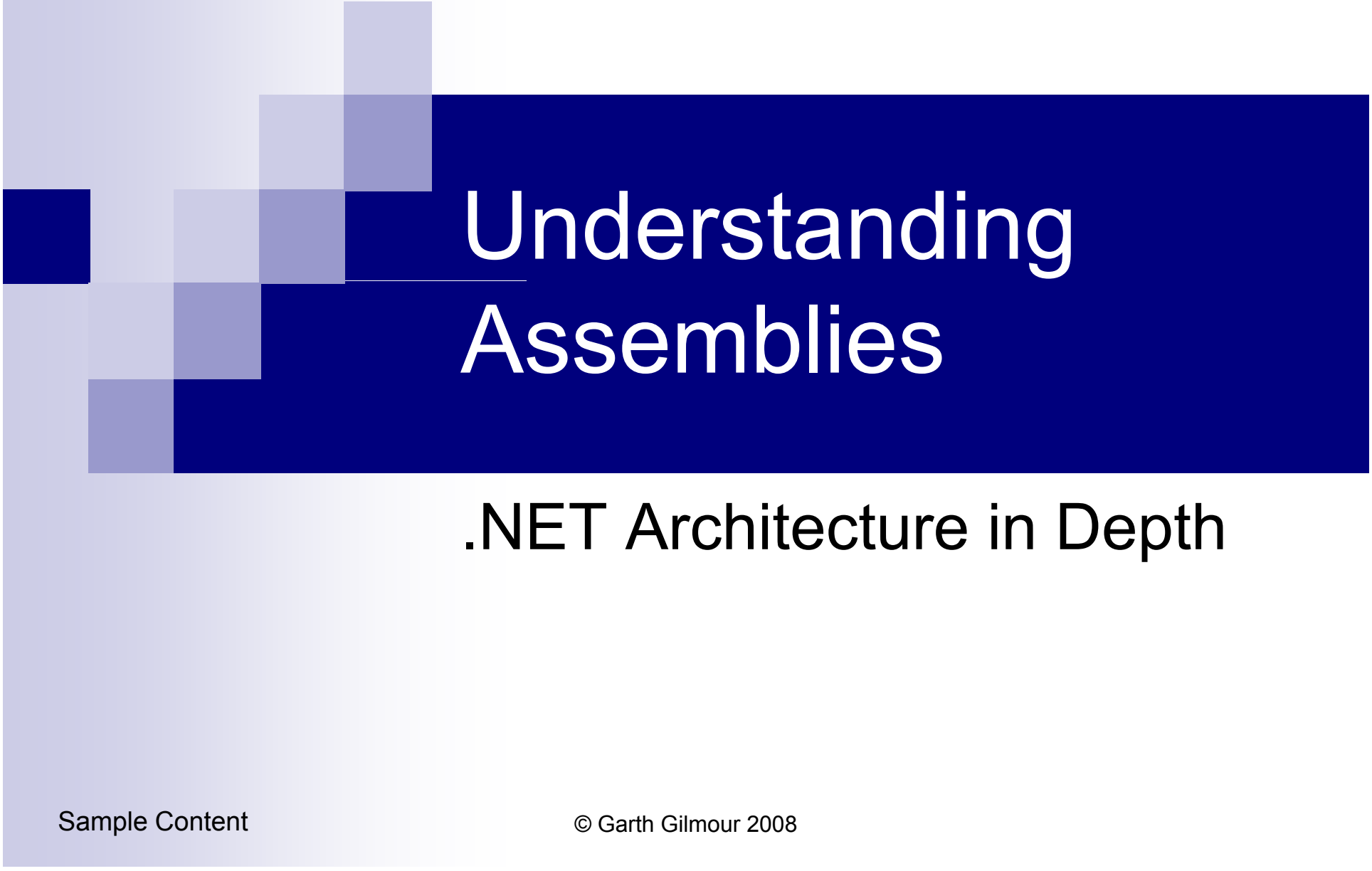


.NET Architecture in Depth

Sample Content

Modules and Assemblies

- A .NET program has three constituents
  - Intermediate Language, Metadata and Resources
  - The compiler creates these as a stream of bytes
- The compiler output is written into a ‘Module’
  - This can be saved as a ‘.netmodule’ file
  - To do this use the ‘/t:module’ compiler option
  - Modules can be compiled separately and combined later
- A Module may contain the ‘initial entry point’
  - The function that is used to launch the program
  - This is specified in IL using the ‘.entrypoint’ directive

The Entry Point Directive

```
.assembly extern mscorlib {}
.assembly Hello {}

.class public auto ansi SayHello extends [mscorlib]System.Object {
      .method public static void FooBar() il managed {
            //entry point declaration (one per assembly)
            .entrypoint
            //load a string onto the execution stack
            ldstr "Hello There"
            //call a method from another assembly
            call void [mscorlib]System.Console::WriteLine(string)
            //return from this method
            ret
      }
}
```

Modules and Assemblies

- An Assembly is a collection of Modules
  - These use the NT Portable Executable file format
- A ‘Managed PE’ file contains
  - The CLR header
    - Which version of the CLR this program targets
  - The metadata
    - Tables describing the assembly and its dependencies
  - The Intermediate Language
    - Stored in a section of the PE previously kept for text
- Individual Modules can be loaded from an Assembly
  - Useful if you are loading types over a network

Types of Assembly

- Command line executables
  - A program which runs in a console window
  - Built by using the '/t:exe' compiler option
- GUI based executables
  - A progam which creates one or more windows
  - Built by using the '/t:winexe' compiler option
- Dynamic libraries
  - A repository for types to be used by the program(s)
  - Built by using the '/t:library' compiler option
  - A library cannot be started directly
    - Visual Studio lets you specify a program for debugging
    - See 'Project→Properites→Configuration Properties→Debugging'

Modules and Assemblies

- An Assembly can contain multiple Modules which were written using more than one .NET language
  - In practice this option is rarely used
  - An Assembly usually maps to a single Module
- Assemblies are often built from different languages
  - Interoperability is guaranteed via the Common Type System
- Assemblies can be dynamically generated
  - At the low level via the types of ‘System.Reflection.Emit’
  - At a high level via the CodeDOM library (which builds an AST)
    - This is very useful for creating templates and code wizards

Naming Assemblies

- .NET assemblies are strongly named
  - They are not located based solely on a filename
- An assembly name is built from four elements
  - The filename of the assembly
  - A four part version number
  - The country/language code
  - The public key of the company
- Only the assembly filename is mandatory
  - There are consequences for leaving other parts out
- Assembly names can be discovered in code
  - Calling ‘Assembly.GetName’ returns an ‘AssemblyName’ object

Names and Strong Names

- The name part of a strong name is the filename
  - Minus the file extension (usually '.dll' or '.exe')
  - This is automatically set by the compiler
- This is sometimes referred to as the 'friendly name'
  - Used in isolation these always lead to deployment problems
- The framework names libraries after namespaces
  - E.g. the classes for manipulating XML live in the namespace 'System.Xml' which is in turn packaged into 'System.Xml.dll'
  - This is a simple namespace-to-module-to-assembly mapping
    - Although it takes some time to get use to filenames with periods
  - Whether this makes sense depends entirely on your design

Versioning and Strong Names

- An Assembly version number has four parts
  - It is expressed as ‘Major.Minor.Build.Revision’
  - If not set the version number is ‘0.0.0.0’
- This is set in code using ‘AssemblyVersionAttribute’
  - E.g. ‘[assembly:AssemblyVersion(“1.2.34.567")] ’
  - Only the major build number is mandatory
    - Omitting any of the other parts causes them to be set to zero
  - The build and revision numbers can both be specified as ‘*’
    - The build no is set to the number of days since ‘01/02/2000’
    - The revision no is set to half the no of seconds since midnight

Culture and Strong Names

- Assemblies holding IL never have a Culture Code
  - Instead they are said to be ‘culture-neutral’
- Only Assemblies holding resources have a Culture Code
  - These are called ‘Resource-Only’ or ‘Satellite’ Assemblies
- The culture is set via ‘AssemblyCultureAttribute’
  - The code is in RFC 1766 format (e.g. ‘en-US’)
  - Each version of your application will have may different copies of the satellite assemblies, one for each locale you support

Encryption and Strong Names

- Assemblies can be digitally signed
  - By generating hash values from their content
- Signed assemblies must include a public key
  - Which is used to verify that hash values are authentic
- Usually only a ‘Public Key Token’ is used in the name
  - This is an 8 byte hash value of the key
  - The key itself is 128 bytes in length
  - In theory this is bad for security
- Keys are the only thing that guarantees uniqueness
  - Vendors could easily duplicate the name, version and culture
    - Such as ‘utils.dll’ version 1.1 or ‘msgs.dll’ with culture code en-US

Describing Strong Names

- A Strong Name can be specified as a string
  - This is for use in code or configuration files
- The string is comma separated with the filename followed by ‘name=value’ pairs for the other parts
  - Because as we have seen only the name is mandatory
- A string with items omitted is ‘partially specified’
  - Any values for the ommitted parts are acceptable
- Culture can be set to ‘Neutral’ and public key to ‘Null’
  - This is NOT the same as a ‘partially specified’ string

```
MyLib, Version=1.2
MyLib, Version=1.2, Culture=en-US, PublicKeyToken=123456789
MyLib, Version=1.2, Culture=Neutral, PublicKeyToken=Null
```

Describing Strong Names in Code

```
class Tester {
      static void Main(string[] args) {
            //Get the application domain for the current thread
            AppDomain current = AppDomain.CurrentDomain;
            //Add a new handler for load events
            current.AssemblyLoad += new AssemblyLoadEventHandler(loadEventHandler);
            //Load an assembly into the current domain
            Assembly assembly = current.Load("System.Xml,
                                                Version=1.0.5000.0,
                                                Culture=Neutral,
                                                PublicKeyToken=b77a5c561934e089");
      }
      //Triggered when we load a new assembly
      static void loadEventHandler(object sender, AssemblyLoadEventArgs args) {
            Console.WriteLine("Just Loaded {0}",args.LoadedAssembly.GetName().Name);
      }
}
```

Assembly Dependencies

- Every assembly stores its metadata in a ‘Manifest’
  - This can be examined using the ‘ildasm’ tool
- Two types of information are stored in the ‘Manifest’
  - Information about the current Assembly and its Modules
  - Descriptions of external Assemblies that are required
- Dependencies can use all four parts of the strong name
  - This can be altered by information in configuration files
- A reference to ‘mscorlib’ is at the top of every manifest
  - Because this is the Assembly in which the CTS resides

External Dependencies

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                    // .z\V.4..
  .ver 1:0:3300:0
}
.assembly extern System
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                    // .z\V.4..
  .ver 1:0:3300:0
}
.assembly extern System.Data
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                    // .z\V.4..
  .ver 1:0:3300:0
}
```

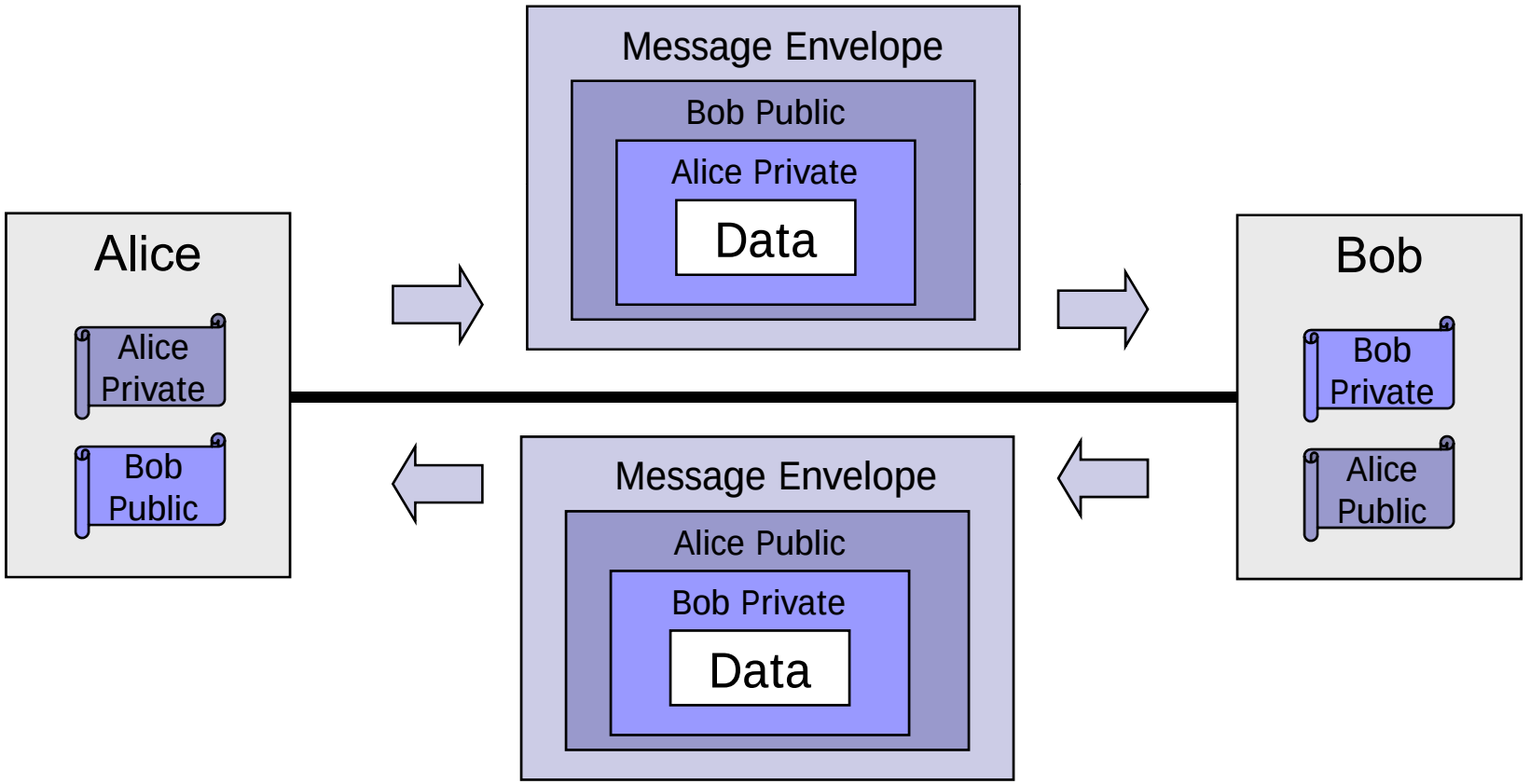
The Current Assembly

```
.assembly BuildingDataSets
{
  // --- The following custom attribute is added automatically, do not uncomment -------
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(bool,
  //                                                                                 bool) = ( 01 00 01 01 00 00 )
  .hash algorithm 0x00008004
  .ver 1:0:1617:32277
}
.module BuildingDataSets.exe
// MVID: {2CC30E23-4B5D-4D24-A117-A99ED07960AA}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 4096
.corflags 0x00000001
// Image base: 0x06c40000
```

Digital Signatures

- .NET relies on asymmetric cryptography
  - More commonly described as public/private key pairs
  - The public key is distributed and the private key secured
- Data encrypted with the private key can only be decrypted with the public key
  - This lets Alice authenticate that a message comes from Bob
- Data encrypted with the public key can only be decrypted with the private key
  - This enables Alice to securely send message to Bob
- The procedures can be combined
  - Bob might sign a message with his private key and then encrypt it using his copy of Alice's public key

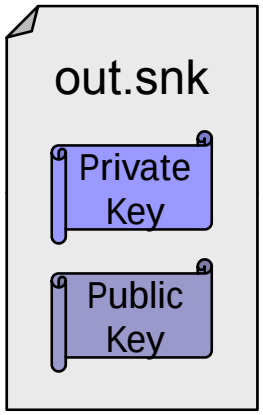
Signing and Encrypting Messages

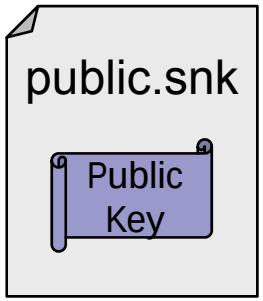


Generating Keys and Hashing Data

- The ‘sn.exe’ tool generates keys
  - ‘sn -k out.snk’ generates a file containing a new key pair
  - ‘sn -p out.snk public.snk’ makes a copy of the public key and places it in a separate file
- The key file is specified to the compiler by using the ‘AssemblyKeyFile’ attribute
  - The private key will be used to sign the Assembly
  - The public key will be used to generate the Public Key Token
- Signing an Assembly can be postponed
  - Developers can use the ‘AssemblyDelaySign’ attribute to leave space for the signature and hence only require the public key
  - The keyholder can add the signature at a later time via ‘sn -R’

Generating Keys and Hashing Data





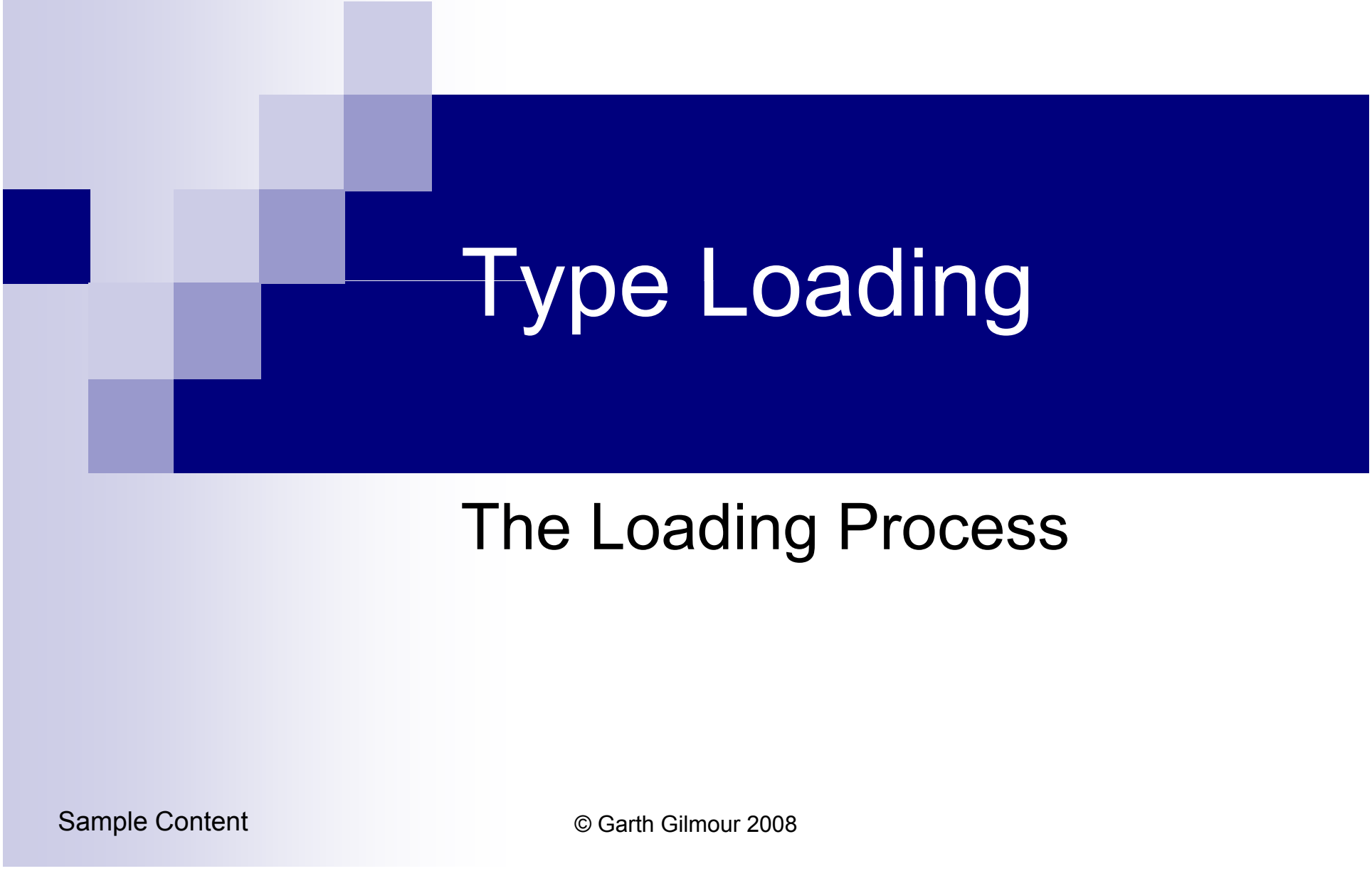
Attaching a Signature to an Assembly

```
[assembly:AssemblyKeyFile("..\\..\\out.snk")]
[assembly:AssemblyDelaySign(false)]
```

Delay Signing an Assembly

```
[assembly:AssemblyKeyFile("..\\..\\public.snk")]
[assembly:AssemblyDelaySign(true)]
```

```
sn -R MyApp.exe out.snk
```



The Loading Process

Loading Types Into the CLR

- The ‘CLR Loader’ loads types into the virtual machine
  - It is up to the deployer to ensure that the strong names of assemblies can be mapped to physical files
- Types are usually loaded on first use
  - ‘Just In Time’ compilation means that unloaded types only need to be resolved when the program first references them
- Resolving an assembly is a well defined process
  - A series of locations are each checked in turn
  - The first matching assembly will be loaded into the CLR

Loading Types Via the DEVPATH

- This is a hack that was designed to ease development
  - It is deprecated in version 2 of the framework so avoid it…
- A ‘DEVPATH’ environment variable can be specified
  - This specifies a folder to be checked for Assemblies before all other locations (the path must end with a slash)
  - This is an easy way to test:
    - Libraries shared by multiple programs
    - Delay-signed assemblies before signing
- This feature must be explicitly enabled
  - In the machine wide configuration file

Loading Types Via the GAC

- In a production environment the GAC is searched first
  - The ‘Global Assembly Cache’ is a repository of Assemblies
  - Assemblies are placed there to be accessed by any other application running on the current machine
- The GAC is only searched if the target assembly is strongly named (including the digital signature)
  - Only strongly named applications can be placed in the GAC to avoid corrupting the cache with multiple implementations
  - The reference name must be complete for the GAC to be used
- The signature is validated when an Assembly is added
  - This removes the need to perform that check at runtime
    - Which provides a small efficiency on each Assembly load

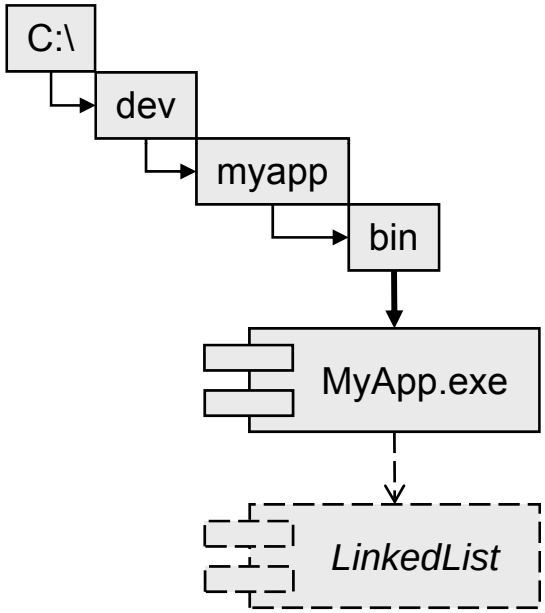
Loading Types Via the Code Base

- The next location to be searched for an Assembly is determined by code base hints
  - These are mappings found in the machine and application specific configuration files
  - The name of an assembly is mapped to a file path or URL
  - Each version of the assembly can have its own mapping
- Once again the assemblies must be strongly named
  - So that an exact match can be made
- Assemblies downloaded over the network are cached
  - There will be a separate cache for each user
  - The assemblies count as ‘mobile code’ and therefore are run with limited permissions

Loading Types Via Probing

- The final location searched for Assemblies is the installation directory and its subfolders
  - This is referred to as the ‘APPBASE’ directory
  - The process of searching it is referred to as ‘probing’
- Assemblies in the current directory will always be found
  - Those in subfolders will be found if
    - The subfolder has the same name as the Assembly
    - The folder is enabled for probing in the config file
- Extra probing is done for Assemblies with a culture code
  - If the library ‘msgs.dll’ has a culture code of ‘en-US’ then the loader will look for an ‘en-US’ folder in the ‘APPBASE’ directory
  - This makes it convenient to bundle multiple Satellite Assemblies

Loading Types Via Probing

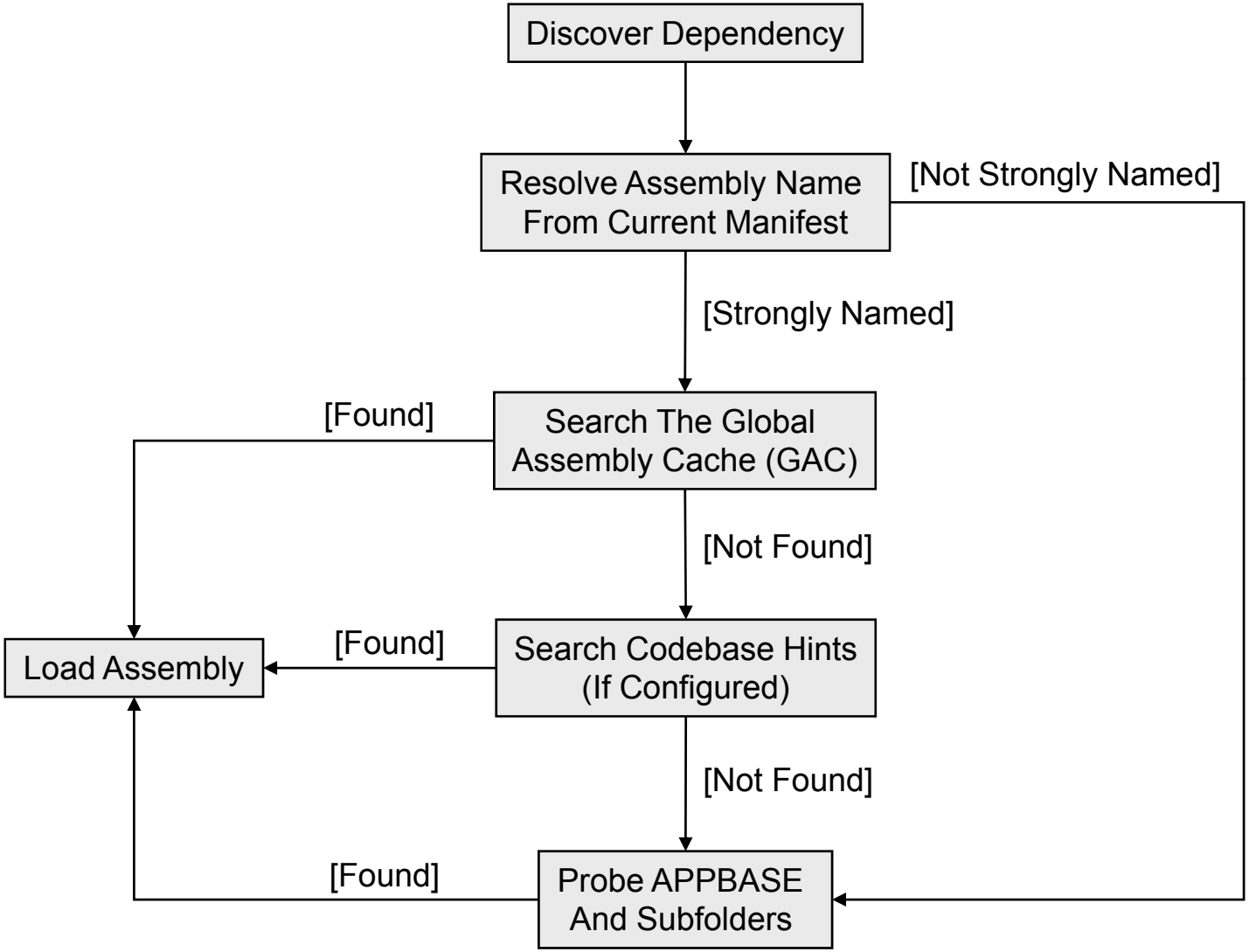


Culture Neutral Probing

```
C:\dev\myapp\bin\LinkedList.dll
C:\dev\myapp\bin\LinkedList\LinkedList.dll
C:\dev\myapp\bin\LinkedList.exe
C:\dev\myapp\bin\LinkedList\LinkedList.dll
```

Culture Dependent Probing

```
C:\dev\myapp\bin\en-US\LinkedList.dll
C:\dev\myapp\bin\en-US\LinkedList\LinkedList.dll
C:\dev\myapp\bin\en-US\LinkedList.exe
C:\dev\myapp\bin\en-US\LinkedList\LinkedList.dll
```

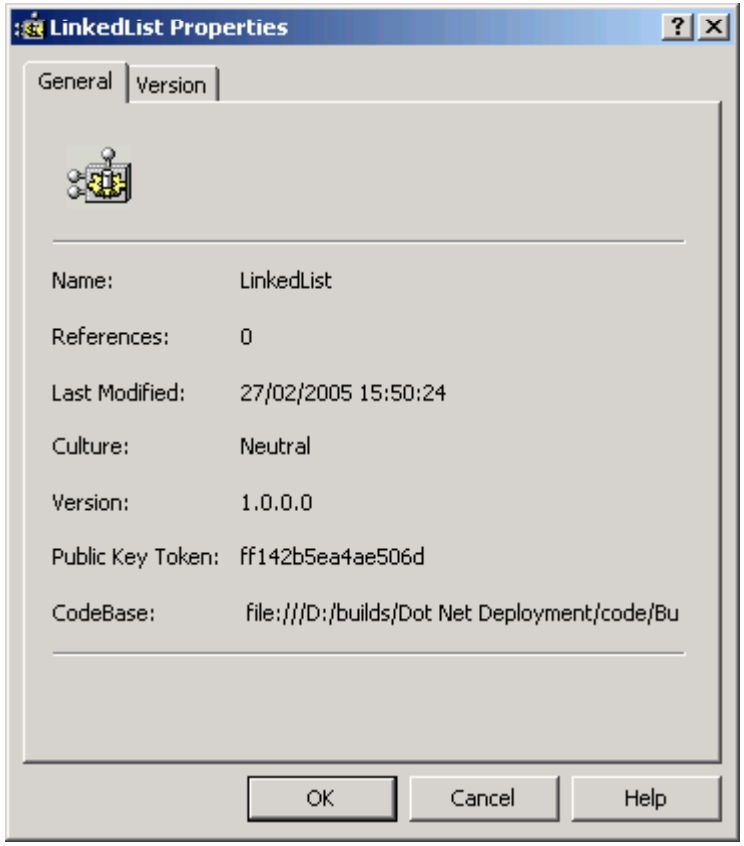


The GAC in Detail

- The implementation if the GAC is irrelevant
  - You should not write code that depends on it because its structure is likely to evolve with the framework
- The GAC is structured as a hierarchy of directories
  - Unique directory paths are built by naming (sub)folders after the four parts of the Assemblies strong name
- This directory structure cannot be used directly
  - It is maintained by a component loaded from 'FUSION.dll'
  - The shell extension 'SHFUSION.DLL' enables you to view the contents of the GAC via Windows Explorer
    - It is automatically triggered with you navigate to the top folder
    - This is '<drive>\Windows\Assembly' on Windows XP machines

The GAC In Detail

- The four components of the libraries ‘Strong Name’
  - The key value is hashed
- The number of programs which depend on the Assembly
  - Used to prevent MSI from unloading a library still in use
- The physical path from which the Assembly was loaded
  - Not visible if the library was loaded from an archive (MSI)



The GAC In Detail

- Many different versions of an Assembly can coexist
  - As long as their strong names differ (usually by version number)
- Assemblies are deployed and/or removed from the GAC using the Global Assembly Cache Tool (gacutil.exe)
  - Care must be taken when removing Assemblies that you only remove the version you want to (so always specify the full name)

Command	Description
gacutil /i MyLib.dll	Install MyLib.dll
gacutil /u MyLib	Uninstall all copies of MyLib.dll
gacutil /u MyLib, Version=2.6, PublicKeyToken=123ABC4	Uninstall just the specified assembly
gacutil /il assemblies.txt	Install using assembly names from file
gacutil /ul assemblies.txt	Uninstall using assembly names from file

.NET Configuration Files

- All .NET configuration files have the same structure
  - They are written as XML documents
  - The document element is called ‘<configuration>’
  - A number of sections can appear in the file
- The file format is extensible
  - You can add extra sections to read in ‘magic numbers’
- The main configuration files are:
  - The machine wide configuration file
  - The applications own configuration file
    - This is slightly different for Web Applications

The Machine Wide Config File

- There is a single ‘machine.config’ file located in the installation directory of the .NET framework
  - This is usually ‘C:\Windows\Microsoft.NET*xxx*\config’
  - Where *xxx* is the version number of the framework
- The ‘machine.config’ file contains the default settings for all aspects of each type of application
  - Including which version of the CLR to use, where to locate assemblies and the locations of remote objects
  - None of these settings are transferred when you distribute an application by copying to another machine
  - As it is used by all applications it should be edited with care

The ‘machine.config’ File

```
<configuration>
    <configSections>
        <!-- Define processor classes for other sections in the file -->
    </configSections>
    <appSettings>
        <!-- Contains application specific name/value pairs-->
    </appSettings>
    <system.diagnostics>
        <!-- Specifies error handlers and levels of error tracing -->
    </system.diagnostics>
    <system.net>
        <!-- Specifies settings for network connections -->
    </system.net>
    <system.web>
        <!-- Specifies ASP.NET specific settings -->
    </system.web>
    <system.runtime.remoting>
        <!-- Specifies remote objects and channels -->
    </system.runtime.remoting>
</configuration>
```

The Application Configuration File

- Each application can have its own configuration file
  - This must be found in the same directory as the application and have the same name plus '.config'
    - So for example 'MyApp.exe' would use 'MyApp.exe.config'
- ASP .NET Web Applications are slightly different
  - The configuration file is always called 'web.config'
  - Settings specific to Web Apps go inside '<system.web>'
- A Web App can have a 'web.config' file in every directory
  - Usually there is a single file which is placed in the base directory
  - Files in subdirectories can override settings from parent files

Type Loading and Versioning

- References to dependant assemblies are exact
  - If you have a reference to version 1.0.0.0 of a library you will not automatically use version 1.1.0.0
- This complicates upgrades and maintenance
  - If client applications have referenced outdated versions of your libraries and you cannot ask them to recompile their code
- One solution is to use version policies
  - These are mappings which redirect an application from an older version of a library to a newer one
- Version policies are specified in the configuration files
  - You can also package the information into a separate DLL and deploying it into the GAC as a ‘Publisher Policy Assembly’

Versioning Assemblies