Plan

- managed modules
- execution of managed code
- metadata
- deployment and assemblies

managed modules

- code is compiled in managed modules
- PE executables
- contain headers (PE and CLR)
- metafata
- IL code

the pe and clr headers

- PE header
  - PE32 or PE32+ format
  - gui, cui or dll
  - timestamp
  - ignored for IL only modules
- CLR header
  - required version of CLR
  - flags
  - MethodRef for entry point
  - location and size of metadata, resources
  - strong name

metadata overview

- set of data tables describing what is defined in the module
- additional information about what the module references
- all metadata is always associated (embedded in) with the module (unlike IDL or TLB)
- uses of metadata
    - no headers and library files needed
    - Visual Studio uses md for IntelliSesnse
    - CLR verification process
    - serialization
    - garbage collection

IL overview

- stack based high level (object-oriented) assembly language
- CPU-independent, no registers
- object oriented features — can instantiate objects (newobj instruction)
  call virtual methods (callvirt), work with members (ldfld, stfld)
- special purpose instructions for some types — arrays (ldelem)
- type independent arithmetic instructions — add, mul
- instruction for loading and storing (constants, indirect, local variables, arguments), eg. ldstr, ldarg, ldloc, stloc
- branching, labels, exceptions handling

modules and assemblies overview

- CLE actually works with assemblies
- assemblies are assembled from one or more module files
- assembly is what we would call a component

loading the CLR

- CLR is loaded by the so called runtime host (native process)
- typically — the windows shell, ASP.NET, Intenet Explorer
- there is an API to load the runtime into a process and run managed code (COR API)
- the runtime must be installed (MSCorEE.dll is present in the system directory)
- version of the runtime — registry, CLRVer utility
  - windows examines the header and creates the appropriate process type
  - windows loads the appropriate version of MSCorEE.dll
  - the primary thread runs the function in MSCorEE.dll, that does initialization, loads the EXE assembly and jumps to the entry point function in it
  - similarly if a process calls LoadLibrary with a dll assembly

executing assembly code

- all types in a method are scanned
- the type tables are created
- when a method is called, the IL is found (using metadata), verified, compiled and stored, the pointer stored in the table
- only one performance hit by the first call
- both the il and native code can/may not be optimized (unoptimized code mainly for debugging — the nops in code)
- why JIT compilation can be faster
  - target platform can be determined at run time (CPU specific instructions)
  - certain tests can be allays false on the target platform
  - JITter could profile the execution of the code and reorganize and the recompile the code

verification

- verification examines the IL code and checks if the code is safe
- it simulates every possible control flow and verifies the stack
- every method is called with correct number of parameters
- that parameters have proper types
- return values are used properly
- based on metadata for methods
- verification allows more applications (AppDomains) to run in one process

unsafe code

- any code containing embedded native code, unmanaged pointers, methods returning managed pointers etc.
- it is not verified — verification is denied or skipped (if the appropriate permission is set)
- PEVerify.exe utility

CTS and CLS quick revision

- the application (assembly) consist of modules
- each consist of types
- types consist of members (fields, methods, properties and events)
- all have visibility — types in the assembly (public or internal) members in the type and assembly (private, protected, protected and/or internal, public)
- CTS defines rules for inheritance, virtual methods, object life-time
- the language code and types behavior are to be considered separate (see C++ multiple inheritance)
- the single root of hierarchy as a basic rule for inheritance
- CLS for language interoperability (some constaints)
- on a very basic level all members are fields and methods

Unmanaged code interoperability

- PInvoke — functions in native dlls can be called directly (using the DLLImport atribute), must define all structures and datatype (StructLayout attribute)
- managed code can use existing COM components (tlbimp utility)
- COM components can use managed code (tlbexp and regasm utilities)
- a very rich topic — marshalling of types etc.

.NET framework design goals

- windows have been considered ”unstable”
- dll hell
- instalation complexity
- security problems

bulding types into modules

- basic command line : `csc /t:exe /r:MsCorLib.dll Program.cs`
- /t switch — module type (exe, winexe, library, module)
- /r switch — referenced assemblies (MsCorLib.dll authomatically)
- common switches in the response file use `csc @respfile file.cs`
- default — local CSC.rsp and global CSC.rep response files

Metadata

- each module contains metadata

definition and reference tables

- definition tables
- ModuleDef, TypeDef, MethodDef, FieldDef, ParamDef, PropertyDef, EventDef
- reference tables
- AssemblyRef, ModuleRef, TypeRef, MemberRef
- use IlDasm to inspect metadata

assemblies

- modules are combined into assemblies, typically one managed module per assembly
- one module is considered primary — it contains special metadata called manifest
- assembly defines reusable types
- assembly is marked with version number
- assembly can have security information associated
- benefits of multimodule assemblies — incremental download, adding resources and datafiles, different prgramming languages in one assembly

Manifest

- one PE file in the assembly contains the assembly metadata, this file is loaded first by the CLR
- AssemblyDef, FileDef, ManifestResourceDef, ExportedTypesDef
- the manifest states that the file is a part of the assembly, the modules do not reference the assembly

AL

- to create assembly use the csc compiler or al assembly linker
- modules are compiled with /t:module switch
- they have .netmodule extensions and are PEs of dll type
- /addmodule switch add a module to the assembly created
- al combines the modules and creates a manifest only module

assemblies and resources

- al /embed or /link switches
- csc /resource and linkresource switches
- /win32res switch

version resource information

- resource information is added to the assembly
- assembly level attributes eg.
  `[assembly:AssemblyFileVersion("1.0.0.0")]`
- AssemblyInfo.cs in Visual Studio
- version resources : AssemblyFileVersion, AssemblyInformationalVersion, AssemblyVersion (relevant to CLR)
- Major version, Minor version, Build number, Revision number

culture

- assemblies containig code should have neutral culture
- satellite assemblies — contain only resources
- use al to create (/embed and /c switches)
- in code use `ResourceManager` object
- use [assembly:AssemblyCulture("en-US")] in code

two kinds of deployment

- private — in one installation directory, weak name (just file name)
- global deployment — assemblies identified by strong name, stored in Global Assembly Cache (strong name)
- global suitable for sharing, violates simple installation goal

strong names and the SN utility

- sn consists of
- file name (without extension)
- version number
- culture
- public key
- sn utility creates a private public key pair : `sn -k file.keys`
- `sn -p keysfile pukeyfile` — use to extract the public key
- `sn -tp pubkeyfile` — use to view public key
- public key token — 64 bits hash of public token
- to sign the assembly use /keyfile switch

the GAC and the Gacutil utility

- GAC path — c:\Windows\Assembly
- the gacutil utility
- gacutil /i — instal assembly
- gacutil /u — uninstall assembly
- gacutil /l — list assemblies

delayed signing

- if you do not have the private key use /delaysign and public key file instead
- use `sn -Vr AssemblyName` so that you can install the assembly in the GAC
- use `sn -R assembly keyfile` to sign the assembly with the private key

resolving type references

- IL refers to a member
- IL refers to a type
- TypeRef indicates ModuleRef, AssemblyRef or ModuleDef
- if ModuleRef or ModuleDef - load the type from the appropriate module (file)
- if AssemblyRef
  - if weakly named - search the AppBase
  - if strongly named - search the GAC and then the AppBase
  - load the manifest file and its ExportedTypesDef

Memory management

- The program resources consume memory
- they are stored on the thread's stack or in the managed heap
- every type is a resource
- the lifetime of a resource
  1. new memory is allocated (newobj)
  2. it is initialized (.ctor)
  3. resource is used by the application
  4. tear down the state (Dispose pattern)
  5. free the memory (Garbage collector)

Advantages over native programming

- no need to worry about the size
- no need to worry about freeing the memory
- so no ugly memory bugs
- but a lot of resources still need to be closed by hand — system handles (files, tokens etc.)

New objects creation

- CLR allocates resources on the managed heap
- it is similar to the C-runtime heap, but it is managed completely by CLR
- when a process is initialized CLR reserves a contiguous re of addresses in the memory
- CLR maintains a pointer (NextObjPtr), initially set to the base address of the region
- when newobj instruction is called, the CLR
  1. calculates the memory required for the types and its base fields
  2. add bytes needed for object overhead (8 bytes for 32 bit 16 bytes for 64 bit environment)
  3. checks if there is enough of free memory on the heap
  4. if so the memory starting the NextObjPtr is zeroed out, the type constructor is called (using NextObjPtr as `this`) the calculated type size is added to NextObjPtr the object address is returned

Advantages over C-runtime heap

- allocating memory means simply adding to a pointer (in C the linked list of records must be walked)
- objects are created in the contiguous manner (in C the consecutively created object can be separated by megabytes of memory
- if you create object with strong relationship consecutively, it can improve performance (FileStream and BinaryWriter)
- but there must be a mechanism to ensure that there is always enough of free space garbage collection

Garbage collection

- a mechanism to find objects no longer needed by the application and reclaim their memory
- is usually executed, when there is not enough memory on the heap (after newobj call the object size + NextObjPtr is an address not in the reserved region)
- if there is not enough memory after the garbage collection ends, exception is thrown (OutOfMemoryException)

Garbage collection

- application has a set of roots storage locations containing a memory pointer to a reference type object (can be null)
- local variables, static fields and method parameters of reference types are roots
- when garbage collection is started it walks the stack determining roots in all the methods tables
- mark phase then it iterates through roots and marks all objects referenced by them (following in-object references recursively, it does not mark objects twice)
- compact phase all not marked objects memory is reclaimed, others are shifted down in memory to keep the heap compact
- all roots references are updated to the shifted addresses
- `NextObjPtr` is updated accordingly

Garbage collection

- performance hit, but occurs only when generation 0 is full
- the lifetime of an object is fully managed by CLR
- no leaks, no access to freed memory
- no memory fragmentation
- the object referenced by the local variable does not live until the end of the method
- in debugged code JIT makes the lifetime longer

Finalization

- last meal for the object before it is killed
- used for freeing the unmanaged resources (file and other operating system handles, network resources)
- when the garbage collector determines that the object is garbage it first calls the method
- the C++ destructor syntax is used ($\sim$`classname`)
- the compiler emits the Finalize method and a try catch block in it that calls the base objects Finalize method

Finalization

- Finalization occurs when
  - generation 0 is full
  - GC.Collect() method is called
  - Windows is reporting low memory
  - CLR unloads an appdomain
  - CLR shuts down
- special thread is used, timeouts are used in some cases (unloading the CLR)
- the GC maintains the Finalization list (objects with Finali method)
- during collection the collected objects from the Finalization list are moved to Freachable queue the references are considered roots
- during the next garbage collection the object are removed from the queue and their Finalize methods are called

the algorithm

- CLR GC is a Generational Garbage Collector
- assumptions:
  - the newer object, the shorter lifetime
  - the older object, the longer lifetime
  - collection a part of the heap is faster then collecting the whole

administrative control and publisher policy

- when the process starts the heap is empty
- the budget size for generation 0 is selected (256 kB CPU L2 cache)
- budget size is selected for generation 1 (say 2 MB)
- budget size is selected for generation 2 (say 10 MB)
- all newly allocated objects are in generation 0
- when the allocation surpasses its size GC is started

algorithm continued

- object that survived are moved to generation 1 only generation 0 is collected until the generation 1 budget size is surpassed
- if generation 1 is full it is also collected and surviving object are moved to generation 2
- only three generations are supported (0,1,2)
- the GC is self tunning (the smaller the budget size, the mor frequent GC)
- e.g. the size of generation 0 can be halved if the lifetime of objects is very short
- if all generation 0 objects are garbage, the memory is freed only by subtracting form NewObjPtr