CLR Confidential talk notes

Basics

* Goals: provide a safe, managed execution environment that is easier for developers to use than unmanaged code, and supports multiple languages and allows interoperation between them. Reduce errors, facilitate library reuse (cf. MFC which you couldn’t use from VB6). Metadata extensibility e.g. MTS/COM+ attributes.
* CLI (the standard) vs. CLR (the implementation). Microsoft CLRs: desktop, Silverlight, WP7 (ARM). Other implementations: Mono, Rotor.
* C# and VB are mostly thin layers over the underlying CLR. Therefore, understanding the CLR is essential to understanding C# and VB. *You cannot use these languages effectively without understanding the platform they run on.*
* Also, many issues and questions that arise turn out to be runtime or framework questions, not language questions. Similarly design guidelines are generally not specific to C# or VB but are about designing for the CLR.
* Therefore, developers need to be conscious of the difference between language issues/idioms, framework issues/idioms and fundamental issues/idioms.
* Three core dimensions: virtual execution system (including services like loading DLLs, memory management, etc.); common type system (enabling language interop); and standard core libraries.

History

* Predecessor: COM. Unmanaged environment with weak and non-extensible metadata support and poor abstraction of runtime services.
* 1.0 and 1.1
* 2.0
* 4.0
* Note 3.0 and 3.5 are not revs of the CLR, only of the higher level frameworks.

Plan of attack

* Two main sections: how the CLR *represents* programs (static aspects), and how the CLR *executes* programs (dynamic aspects)

Static aspects of the CLR

Types

* Fundamental concepts: types and members.
* Every piece of data has a type. It’s not like C where you have a bag of bits (possibly even uninitialized ones!) and can interpret it however you like: instead, the runtime ensures that items are of the correct type.
* The CLR defines a number of types: 8-64 bit integral types in both signed and unsigned variants, 32 and 64 bit floating point types, native int (IntPtr/UIntPtr), char, string, object.
* Other types are defined by libraries (e.g. Decimal in the BCL).
* Reference (managed reference) and pointer (raw pointer) types.
* The CLR type system is object-oriented and unified (all data are convertible to object). Note not all *types* inherit from object e.g. reference/pointer types, interfaces.

What kinds of type are there?

* Reference types – classes
* Value types – structs
* Interface types
* A bit more complicated: managed references and pointers

What is the difference between reference and value types?

* Reference types have *identity*. There can be several references to the same instance.
* Value types have only *value*. When you pass a value, the bits are copied: you get a new instance.
* Objects of reference types are not accessed directly, but through a reference (that is, a variable of reference type stores a reference rather than the object itself). A variable of value type contains the very bits of the object itself.
* Corollary: value types must have a zero value (for array allocations) – or rather, they *do* have a zero value, and custom value types must deal with this.
* Corollary: value types can’t have nondefault field initialisers or otherwise do stuff in their default constructor.
* When should you use a value type? When you have a small object that represents a value.
  + Value types are also useful for small wrapper types e.g. ImmutableArray<T>.
  + Performance implications of large value types – e.g. fields getting reloaded onto the stack
* Idiom: value types should be immutable. (Because when you change a Money value from $1000 to $2000, you’re not changing the value of $1000. You’re specifying a *different* value.)
* Equality semantics: reference types usually have identity equality semantics, value types usually have value equality semantics.
* The curious case of String: value semantics but implemented as reference type. Why? (1) Efficiency in passing strings. (2) Values must be fixed size.

Boxing

* Object is a reference type (so are interfaces). What happens when you store a value type in an object reference? Boxing.
* Unboxing. Note C# ‘cast object to value type’ results in a CLR unbox, not a case – hence uglinesses like (long)(int)obj.
* What happens if you modify a boxed instance? If you modify it through the boxed value (e.g. reflection, dynamic), then it works. [But note that what gets modified is the *boxed copy*, not the original it was copied from.] But if you unbox it then the unbox gets you a new copy, so you won’t modify the boxed instance! It’s confusing – just another argument for immutability.
* Accessing a value type via an interface => boxing so beware!

Generic types

Members

* Type initialisers
* The difference between const and static readonly
* Interface implementations and the multiple “method tables” (one method table but multiple interface sections within it)
* Explicit interface implementation
* Properties are methods
* Specific members deserving consideration:
  + Equals
  + GetHashCode (must remain same over lifetime)

Dynamic (runtime) aspects of the CLR

Managed code

* Virtual Execution System
  + CIL
  + Stack machine
* Framework load
  + SxS vs. in-place Framework updates
  + Mention Project K stuff
* JITter
  + Method table rewriting
  + RyuJIT
  + .NET Native
  + Inlining (why stack traces sometimes leave things out)
  + Tail recursion
  + Reminder: stack traces are not *how you got here*, they are *where you are going back to*
* Type checking
* Null reference checking

How objects are laid out in memory

* Overhead of reference types (type handle and syncblock index) [and that’s why you can’t lock on a struct]
  + On x86 type handle and index are 4 bytes each for total overhead of 8 bytes per object, on x64 8 bytes each for 16 bytes per object
* Atomicity of reads and writes

Garbage collection

* Basic concepts
  + Mark and sweep
  + Eligibility
  + Weak references
* Finalisation
  + Finalisers, finalisation list (put on here at instantiation time) and freachable queue (put on here when collected)
  + CriticalFinaliserObject (finalisers JITted at load time, finalised after non-critical)
  + Resurrection and GC.ReRegisterForFinalize
* Generational GC
* Different iterations of GC for different trade‑offs
* Large object heap

The memory model

* Processors can reorder stuff (e.g. ARM)
* MemoryBarrier and System.Threading.Volatile.Read/Write (“if a read or write appears before this method in the code, the processor cannot move it after this method”).
  + Allegedly C# volatile modifier automatically converts variable accesses to Volatile calls (but can’t be used on array elements) – but actually seems to emit the volatile. prefix on the ldsfld and stsfld op codes… (But I was testing this on Win7… might be different on Win8 especially with PCLs targeting Windows Phone or WinRT.)
* Lazy<T> vs double-checked locking

Assembly loading

* Strong naming

AppDomains

* Isolation boundaries
* Configuration e.g. separate ApplicationBase per domain
* Serialisation vs. remoting (exceptions! And that all‑important serialisation constructor)

Hosting the CLR

* Examples: SQL Server, Windows Runtime

Exceptions

* throw vs throw ex
* Performance?