Introduction to .NET

Part I

We can use .NET to program:

- Console applications
- Windows applications
- Web applications
- XML Web Services
- WPF applications
- Silverlight applications
- Mini-device applications (e.g., smart phones)
- And all of these applications can seamlessly integrate with databases

The Framework

- To grasp .NET programming, one must come to some understanding of its essential components and how they work together.
- We begin by discussing the **.NET Framework**.

The Framework

- Framework consists of two elements:
- a Run time environment called the Common Language Runtime (CLR)
- a class library called the Framework Class Library (FCL)
- The FCL is built on top of the CLR and provides services needed by modern applications.

Together, these two elements provide for:

- Consistent programming model
- Simplified programming model
- Run once, run always
- Simplified deployment
- Wide platform reach
- Programming language integration
- Automatic memory management
- Type-safe verification
- Better Security
- Interoperability

Common Language Runtime

- The CLR underlies everything in .NET
- It is a modern **runtime environment** that
- manages the entire life cycle of an application (is why it's called "managed code")
- It provides services such as:
  - JIT compilation
  - memory management
  - exception management
  - debugging and profiling support
  - and integrated security and permission management

CLR

- The common language specification (CLS) is a set of rules that a language compiler must adhere to in order to create .NET applications that run in the CLR.
- Related to the CLR is *Managed Code* – this is code that's running under the auspices of the CLR and is thus being managed by the CLR.

Framework Class Libraries (FCL)

- The .NET Framework *class libraries* are crucial to program development.
- Thus, C# itself has no class libraries. All languages use the FCL.
- To run applications, the target machine must have the CLR and Framework installed.

Compilation

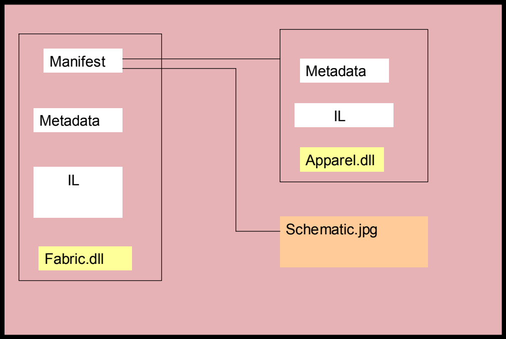
- C# code → Compiler → Managed Module
- Managed Module contains:
- 1. P(ortable) E(xecutable) Header
  - indicates type of file (gui, cui, dll)
  - has timestamp
- 2. CLR Header (72 bytes) – interpreted by run time and other utilities
  - version of CLR
  - location/size of metadata

Compilation

- 3. Metadata
  - Tables that describe types and members defined in your code
  - Tables that describe types and members referenced by your code
- 4. IL Code – will later be compiled by CLR into native machine instructions.

Assemblies

- The CLR doesn't actually work with modules, it works with assemblies.
- All of the managed code that runs in .NET must be contained in an assembly.
- CLR compilers, in fact, take code and other resources needed by the code (jpeg, gif, html, etc) and produce what's known as an **assembly**.
- This is just a logical grouping of one or more managed modules or resource files and is always referenced as one EXE or DLL.



Assemblies

- One file in the assembly must contain what is called a **manifest**.
- The manifest is a set of metadata tables that basically contain
  - the names of all of the files that comprise the assembly
  - the assembly's version
  - culture
  - publicly exported types
- The CLR always operates on assemblies – i.e., it always loads the file that contains the manifest metadata tables first, and then uses the manifest to get the names of the other files that are also in the assembly.

Executing the Code

- Once the IL is created, when an application is scheduled to run, the IL must be converted to native machine instructions.

```
static void Main()
{
  Console.WriteLine("Hello");
  Console.WriteLine("Goodbye");
}
```

Just before Main executes, the CLR detects all the **types** that are referenced by Main's code.

In this case, there is only one type (System.Console).
It has to load the assembly that contains the Console class.
That's mscorlib.dll

In reality it would already be loaded because that also contains Object, the base class for all types in C#

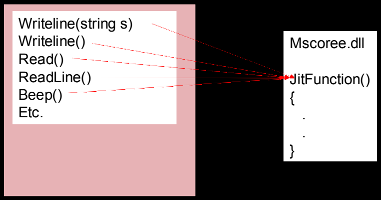
Executing code

- CLR allocates an internal data structure that is used to manage access to the referenced type(s).
- It is an array of function pointers.
- This structure contains an entry for each method defined by the type.
- So each method defined by System.Console has an entry, and each entry holds the address where the method's implementation can be found.
- This is called the **method table**.
- C++ does a similar thing, but only puts in those methods that are virtual.
- In C#, every method gets put in – static virtual, non-virtual, etc.

Executing code

When CLR first initializes this structure, it sets each entry to the address of a method inside mscoree.dll (the execution engine itself for the runtime) that's called the **JITCompiler**.

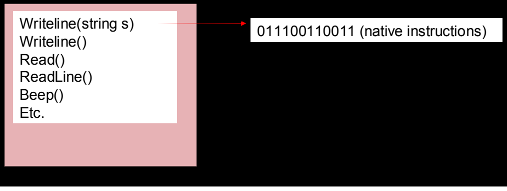
When the thread wants to call WriteLine,
runtime looks up that entry in the data structure (one with a string)
and takes the path to the JitCompiler function.



When WriteLine is called, the JITer searches the defining assembly's metadata for the called method's IL.

It next verifies and compiles the IL into **native instructions** (x86, itanium, opteron) and saves them in a dynamically allocated block of memory.

It then goes back to the internal structure that represents the type, and fills in the address of the dynamically allocated block of code with the native instructions.



Executing code

- Finally it jumps to the code in the memory block and executes it. Then control resumes with the next statement in Main().
- WriteLine again. But this time the code has already been verified and compiled, so the call goes directly to the block of memory, bypassing the JITer.
- So there's a performance hit the first time a method is called. Subsequent calls to method not recompiled.

Notes of the Jiter

- JITer writes **optimized** code (e.g., may know this is a P4 and so can use optimized instructions for that processor)
- Unmanaged compilers (e.g., traditional C++ ones) often have to produce code for the lowest common denominator CPU.
- At runtime, when an assembly is loaded, the CLR automatically checks to see whether there's a **precompiled version** of the assembly, and if so, it loads that one and no compilation is done.

3 Jiters in the SDK

- 1.) Install time code generation – compiles an entire assembly into cpu specific code. Done at install time so entire app is ready to go…no waiting until a function is called.
- 2.) JIT – the default one is called at run time as described above
- 3.) EconoJIT – another run time JITer, designed for hand held devices with limited memory – can discard compiled code (called *code pitching*) when memory gets short. Must recompile if function executed again.

Framework Class Library

- This framework replaces most (not all) of the standard Windows API functionality found since 1985.
- It provides a diverse array of software services, including:
  - Support for core functionality, such as interacting with basic data types and collections, the console, network and file I/O, and interacting with other run time related facilities.
  - Support for interacting with databases and consuming and producing XML
  - Support for building web-based (thin client) applications with a rich server side event model
  - Support for building desktop based (thick client) applications with broad support for Windows GUI
  - Support for building SOAP based XML web services

FCL

- Intermediate Language is not tied to any specific cpu (can be deployed on any machine that supports the framework).
- Best benefit of IL is security and code verification.