CIS 193 – C# Programming

.NET Framework Internals
The C# Compilation Pipeline

C# Sources → Compilation → Common Intermediate Language (CIL) → Assembly (.exe)
The C# Compilation Pipeline (in Full)

- **C# Sources**
- **Assembly references (\*.dll or \*.exe)**
- **Netmodules**
- **Compilation**
  - Common Intermediate Language (CIL)
- **Assembly (\*.exe or \*.dll)**
  - Metadata
  - IL Bytecode
- **Runtime (JIT)**
  - Machine-specific code
Assemblies

• Fundamental unit of deployment in .NET
• Contains metadata and code
• Two forms
  – Executable (.exe)
  – Library (.dll)
Creating .NET Libraries

• A library is a collection of code callable by other entities
  – Allows for logical/practical separation of components (e.g., library + driver)
  – Does not contains a Main method as a result

• From the command-line
  – csc /target:library test.cs

• From Visual Studio
  – Create a class library project
  – Project properties -> Application -> Output type -> 
    Class Library
Using .NET libraries

• At compilation, the library is provided as a reference
  – From the command-line
    • csc /reference:test.dll driver.cs
  – From Visual Studio
    • Right-click References -> Add Reference...

• At runtime, the library must be present as well
  – Complications: versioning, the global assembly cache (GAC), side-by-side (SxS), ...
Assemblies = Metadata + Code

- Metadata includes
  - Assembly manifest (think ship manifest)
    - Version, culture info, files, strong name signature, etc.
  - Type information
    - Common Type System (CTS) format
    - Utilized at compilation time as a reference
    - Utilized at runtime as verification

- Code is... code
  - Common Intermediate Language (CIL) format, formerly called MSIL
The Common Type System

• Essentially C#’s type system
  – “C# is the C of the .NET platform”

```csharp
public class HelloWorld
{
    ...
}
```

```
.class public auto ansi beforefieldinit HelloWorld
   extends [mscorlib]System.Object {
   ...
} // end of class HelloWorld
```
The CTS in Short

• C# is a reflection of the CTS
  – Object-oriented, single-inheritance with generics
  – Distinguished reference and value types

• All languages on the .NET framework compile down to the CTS
  – Even those that are not “like” C# (IronPython, F#)
  – Engineering challenge for language designers on .NET
  – Conceptual challenge for developers who use *multiple languages* on the .NET platform
Reflection

- Reflection is the ability to inspect and operate over the types of objects at runtime.
  - E.g., instantiate an object of a particular type in a satellite library or for testing purposes
- Metadata provides us with the necessary info!
- System.Reflection
  - Classes: Type, Assembly, ConstructorInfo, FieldInfo, MethodInfo, etc.
Common Intermediate Language

• Stack-based IL: locals are pushed/popped
• Good to know it’s there.

```csharp
int x = 10;
Console.WriteLine("Hello World {0}", x);
```

```plaintext
.maxstack 2
.locals init (int32 V_0)
IL_0000:  nop
IL_0001:  ldc.i4.s  10
IL_0003:  stloc.0
IL_0004:  ldstr    "Hello World: {0}"
IL_0004:  ldloc.0
IL_000a:  box      [mscorlib]System.Int32
IL_000f:  call      void [mscorlib]System.Console::WriteLine(string, object)
```
Tools for inspecting assemblies

• ildasm (VS command-line tool)
  – E.g., ildasm test.dll /text

• .NET reflector (freeware)

• (There also exists tools for creating assemblies, e.g., ilasm)
Netmodules

• Assemblies without the manifest
  – I.e., not true units of deployment
• Useful when you need to *separately compile* different components
  – Analogous with .o/.obj files in C
  – E.g., compile a component in C#, another in VB all into the same DLL
• From the command-line
  – `csc /target:module` test.cs
The Final Step: Execution

• Assemblies contain CIL bytecodes
• Two ways to execute bytecodes
  – Create an *interpreter or virtual machine* that reads and executes bytecodes
    • Straight-forward (kind of) but slow
  – Compile bytecodes at *runtime* to machine code
    • Not straight-forward (definitely) but (surprisingly) fast
Just-in-Time Compilation

• .NET bytecode is *jitted* at runtime to machine code.
• Obvious cost at start-up time
• Not-so-obvious performance benefits
  – System-specific optimizations
  – Global optimizations (e.g., inlining of libraries)
    possible without complication or static overhead
  – Profiling-based optimizations
• JIT performs better than interpretation,
  sometimes better than static compilation!
  – (On average...)
JIT Engineering Tradeoffs

• Trade-off between slow start-up and potential optimizations in later executions
  – One approach: do both, e.g., Java HotSpot
    • Initially interpret bytecode, then compile “hot” spots.
  – Another approach: pre-JIT to machine code
    • The ngen utility for .NET
More information

• Overview of the .NET framework (MSDN)

• Compiling MSIL to Native Code