CLR
A new virtual machine

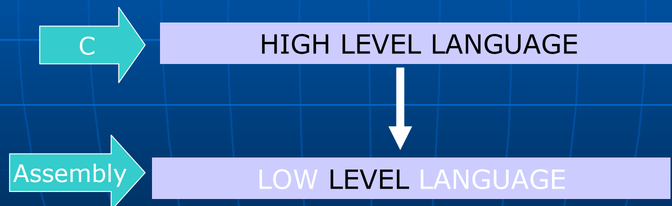
João Ferreira
joao.ferreira@progmat.com

Introduction:

- What is a virtual machine?
- CLI and CLR
- JIT execution and performance on CLR
- Unix with ".NET"

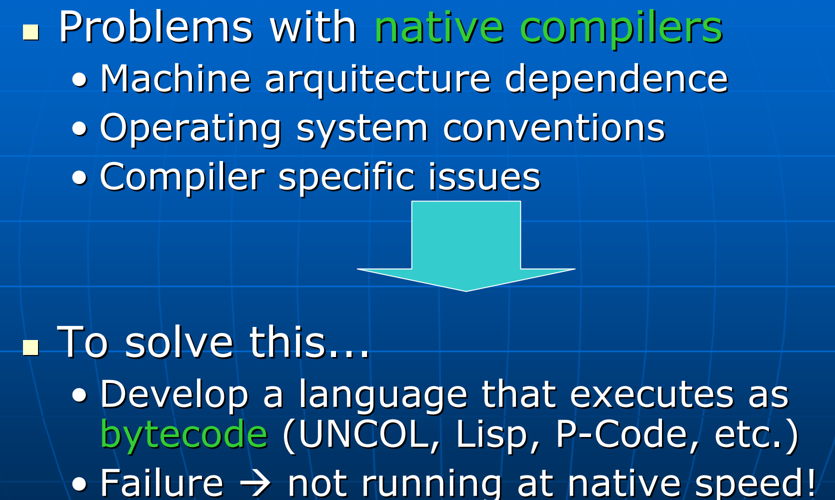
Virtual Machines

- Why "virtual"?
  - Emulation software that translates from one language to another



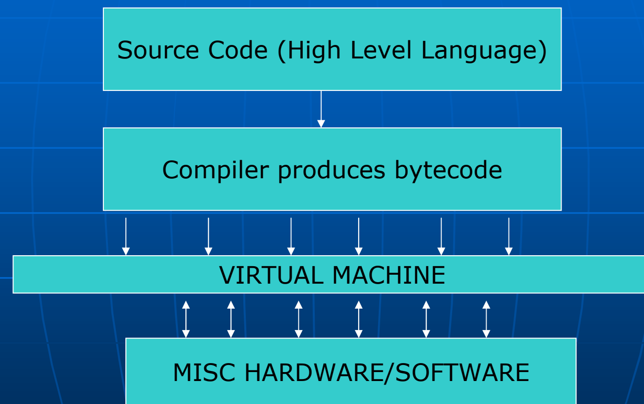
Virtual Machines

- Problems with native compilers
  - Machine arquitecture dependence
  - Operating system conventions
  - Compiler specific issues



- To solve this...
  - Develop a language that executes as bytecode (UNCOL, Lisp, P-Code, etc.)
  - Failure → not running at native speed!

Virtual Machines



New era of VM

- JVM and CLR
  - Use JIT (Just-In-Time) to produce well-optimized native machine code
  - Bytecode (portability)

    +
  - native code generation (speed)

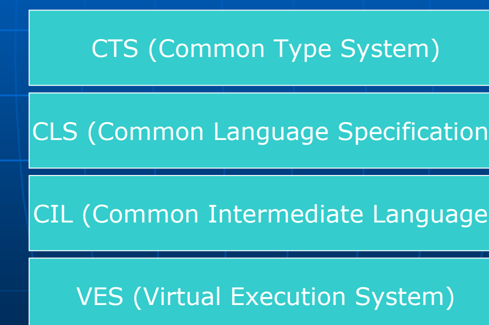
    +
  - Bytecode verifier
  - Stack based – doesn't now the mean of registers

Common Language Runtime

- Microsoft marketing → .NET (DotNet)
- Is nothing more than a virtual machine, however..
- Based on CLI → a standard designed from the scracth to support multiple languages

Common Language Infraestructure

- International Standard (ECMA)
- CLI Specification



CTS (Common Type Specification)

- A rich type system that supports the types and operations found in many programming languages
- To support a wide range of programming languages
- Defines a set of rules for types
- Designed for object oriented, procedural and functional languages
- More than 15 languages implemented CTS

CLS (Common Language Specification)

- A subset of CTS
- Defines the rules for each individual language interop with each other

CIL (Common Intermediate Language)

- Also known as IL or MSIL (Microsoft)
- A language more "higher" than native instruction set
- Based on metadata – self describing
- An assembly oriented language
  - Can create instances of objects
  - Call virtual methods
  - Work with arrays
  - Throw and catch exceptions!

CIL Example

```
public static void TestMethod (int a, int b) // arguments
{
        int c; int d; int e; // locals
        c = a + b;
        d = 10;
        e = c + d;
}
```

ldarg *num* load argument no. num onto the stack

… → …, value

ldarg.1

ldloc *indx* load local variable no. indx onto the stack

… → …, value

ldloc.1

CIL Example

ldc *num* — load numeric constant

… → …, num

ldc.i4 10

stloc *indx* — pop value from stack to local variable

…, value → …

stloc.0

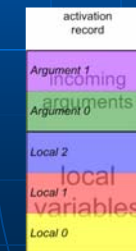
starg *num* — store a value in an argument slot

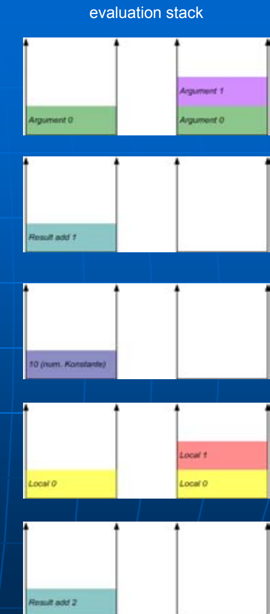
…, value → …

starg.1


```
public static void TestMethod (int a, int b)
{
        int c; int d; int e;
        c = a + b;
        d = 10;
        e = c + d;
}
```

```
.method public hidebysig static void  TestMethod(int32 a,int32
b) cil managed
{
  // Code size       12 (0xc)
  .maxstack  2
  .locals init ([0] int32 c,
           [1] int32 d,
           [2] int32 e)
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  add
  IL_0003:  stloc.0
  IL_0004:  ldc.i4.s   10
  IL_0006:  stloc.1
  IL_0007:  ldloc.0
  IL_0008:  ldloc.1
  IL_0009:  add
  IL_000a:  stloc.2
  IL_000b:  ret
} // end of method Class1::TestMethod
```





VES (Virtual Execution System)

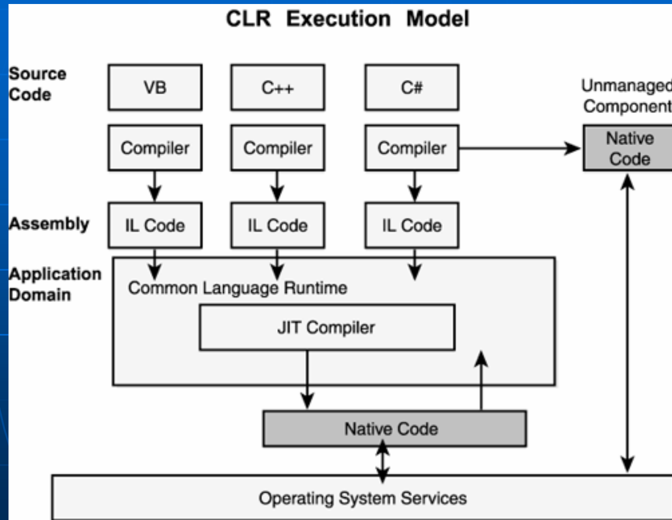
- Implements and enforces CTS
- Will load and run programs written for the CLI (with IL code)
- Handles all the major overheads of traditional programming models
- How it will load?
  - JIT
  - Install time compilers

CLR

- Is the Microsoft VES (Virtual Execution System) for CLI
- CLR will use:
  - Managed modules
  - Assemblies
  - JIT compilers

To run CLI-compliant programms into Win32 and Intel x86 architectures.

CLR Execution Model



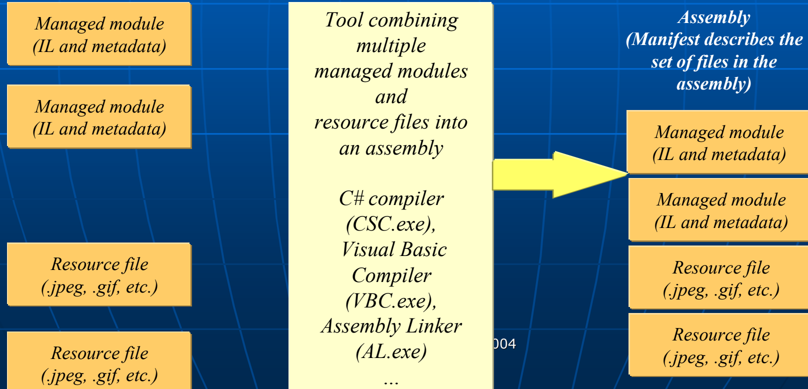
During Development...

- Ok.. Some code...written with notepad:

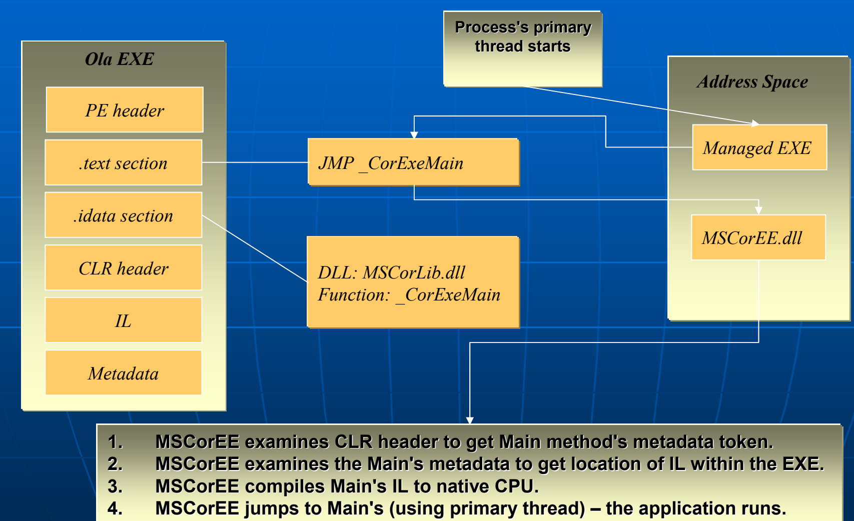
```
using System;
class Hello
{
    public static void Main()
    {
        System.Console.WriteLine("Hello world!");
        System.Console.WriteLine("Bye world!");
    }
}
```


Compiling the source code

- csc ola.il (managed code)
- Result: ola.exe → **Assembly!**
- Resulting file is a PE (Portable Executable)



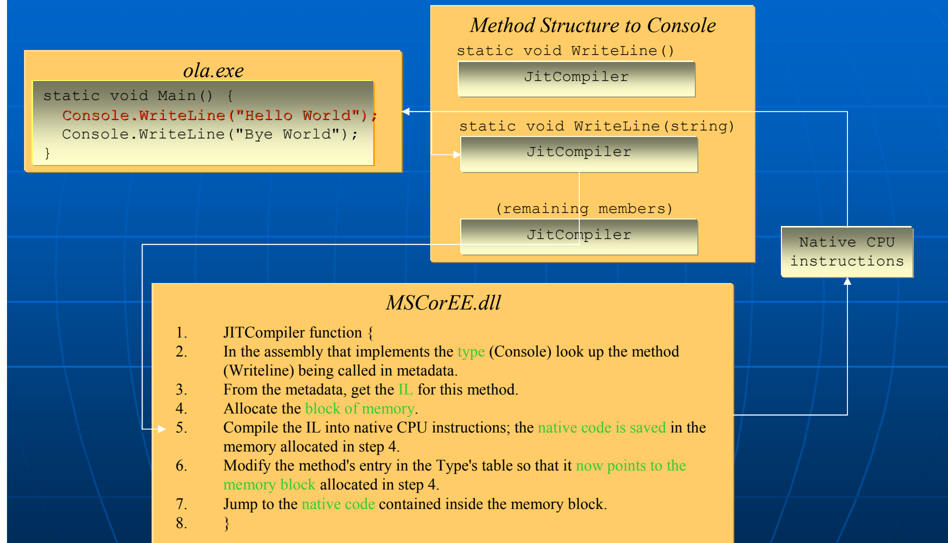
User executes ola.exe... CLR starts!



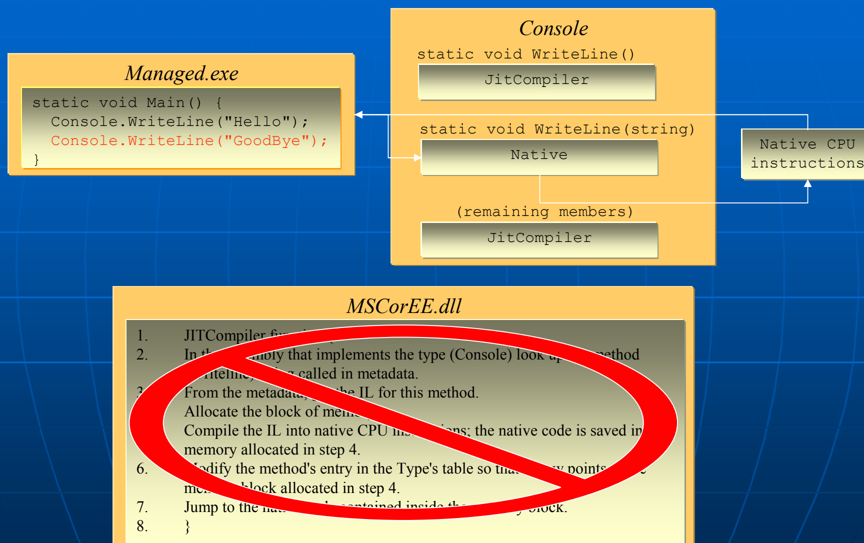
Ok.. We got the IL.. And now??

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       21 (0x15)
  .maxstack  1
IL_0000:  ldstr   "Hello world!"
IL_0005:  call  void [mscorlib]System.Console::WriteLine(string)
IL_000a:  ldstr       "Bye world!"
IL_000f: call void [mscorlib]System.Console::WriteLine(string)
 IL_0014:  ret
} // end of method Hello::Main
```


JIT will translate IL to CPU instructions



Already Jited?



JIT Performance and Issues

- Platform Independence
  - Realized when high-level language compilers convert source code to platform agnostic MSIL code
  - The application or software component is distributed in this form
  - JIT compiles to native code either at runtime or at install time

JIT Performance and Issues

- Language Interoperability
  - Occurs when different language compilers compile to language-agnostic MSIL code
  - Metadata and the Common Type System play a major role in cross-language and platform independence

JIT Performance and Issues

- Runtime Stack Manipulation
  - The JIT Compiler populates important data structures for object tracking and specific stack-frame construction
  - The JIT Compiler can be used to identify specific code elements as they are consumed, i.e., exception handlers and security descriptors (Verifier)
  - Doesn't need registers
  - CPU independent

JIT performance and issues

- Small Memory Footprint
  - JIT compilation takes advantage of the possibility that some code may never be used
  - The JIT Compiler compiles methods only as needed

JIT Performance and Issues

- JIT compiler knows more about the execution environment than an unmanaged compiler would know
- JIT compiler can take advantage of instructions offered by the chip that the unmanaged compiler knows nothing about
- JIT compiler could detect that a certain test is always false, and short-circuit
- The CLR could profile the code's execution and recompile the IL on the fly reducing branching, etc.

The problems...

- Unmanaged code is pre-compiled and can just execute
- Managed code requires 2 compilation phases
  - Compiler produces IL
  - IL compiled to native code at runtime, requiring more memory to be allocated, and additional CPU cycles

Alternatives to JIT

- NGEN.EXE – Install Time tool to create a native image into the native cache
- Native Image → a file containing compiled processor-specific machine code
- Good for heavy startup applications
- Use it at client-side

Unix with .NET

Mono Project

www.go-mono.com

- DotGnu
- http://www.gnu.org/projects/dotgnu

What they did?

- Designed a compiler generating IL code (ECMA 334) – C# taking note of CTS and CLS rules
- A VES with JIT's
- A base class library resides in the `mcs' module in the directoy `class'. Each directory in the directory represents the assembly where the code belongs to, and inside each directory they divide the code based on the namespace they implement.

DotGnu

- Created ilrun → interpret programms in the CIL bytecode format (Ecma 335)
- CIL → converted to CVM (Converted Virtual Machine)
- Cscc – compiler ANSI C and Ecma 334

Conclusion

- CLI is a standard – if you want portability, all languages should be CLI-compliant
- Want run CLI?
  - make your own VES!
  - Make your JIT!