

# **Advanced .NET Programming (CLR Internals and Tools)**

**Tomáš Matoušek**

[tm.matfyz.cz/teaching/clr.htm](http://tm.matfyz.cz/teaching/clr.htm)

# Who Is This Guy?

- 1<sup>st</sup> year PhD student
  - specialization: Software Verification, Static Analysis
- project Phalanger
  - a compiler of the PHP language for .NET Framework
  - final version 1.0 released on Monday :-)
  - <http://php-compiler.net>
- hired to Microsoft CLR Team
  - starting in October in Dynamic Languages Team
- far from know-all
  - your comments and feedback welcomed

# What Is the Seminar About?

- prerequisites
  - .NET Framework Principles (PRG035)
  - winter semester
  - you should know C# and basic .NET principles well
- this semester
  - advanced topics
  - two sessions
    - Pavel Jezek: Tuesday, 17:20, S1
    - Tomas Matousek: Thursday, 15:40, SW2

## **Tuesday's Session, 17:20, S1**

- ADO.NET
- Web Services
- WinForms
- C# 3.0
- Managed C++, C++/CLI
- Appdomains & assembly loading
- Inside .NET objects, strings and arrays
- Garbage Collector
- Security
- ASP.NET
- COM Interop

# Thursday's Session, 15:40, SW2

- CLI implementations
  - Rotor (SSCLI) and Mono
- run-time internals
  - assembly binding, type loading
  - generics
  - native/managed code interop
  - just-in-time compiler
- compiler related
  - compiler tools (Bison equivalents on .NET)
  - API for metadata reading and generation
  - IL assembler, emitting IL code
  - Lightweight Code Generation (LCG)
- CLR hosting (SQL server, Internet Explorer)
- how to write
  - .NET debugger
  - .NET profiler
  - Visual Studio Integration package
- how tools work
  - NGEN
  - MSBuild

# CLI Implementations

- **Microsoft .NET Framework**
  - versions: 1.0, 1.1, 2.0
  - integrated with Windows Vista
- **The Compact Framework**
  - light weighted version of FW for PDAs, mobile phones, ...
  - Windows CE
- **Mono**
  - open source
  - platforms: Windows, Linux
- **Rotor**
  - a part of FW 1.0 released for research
  - source code available
  - platforms: Windows, FreeBSD
  - v2.0 will be available in a few months

# Basic Terminology

- managed vs. native code
- managed vs. native data
- evaluation vs. native stack

# Managed Code vs. Native Code

- code
  - managed vs. native (unmanaged)
  - managed code – IL instructions
  - native code – CPU instructions
  - interoperability (one can call the other)
  - managed code jitted to native at runtime
- Common Intermediate Language (CIL, MSIL, IL)
  - assembler for managed code
  - stack based
  - high level instructions (e.g. newobj, ldstr, throw, callvirt, ...)
  - literals: strings, integers, reals, booleans, null
  - local variables, arguments, fields, methods, functions

# Managed Data vs. Native Data

- managed data (managed heap)
  - managed by Garbage Collector
- native data (native heap)
  - unknown for GC
  - explicit release and destruction

# Evaluation Stack vs. Native Stack

- evaluation stack
  - a stack of the virtual machine
  - each slot can contain item of arbitrary size
    - like Turing machine tape cell
    - a type of data stored in the slot is always known
  - IL instructions add/remove top items of the stack
- native stack
  - real stack, hidden to IL instructions
  - stack frames
    - arguments of methods
    - local variables
    - reflection information
  - code can reflect the current stack and see the callers
  - optimizations by JITter (inlining)

# Introduction to SSCLI – Rotor

- one of the CLI implementations
- source code available
- interesting source code directories:
  - clr/src
    - /vm – Execution Engine (C++)
    - /fjit – JITter (C++)
    - /csharp – C# compiler (C++)
    - /ilasm – IL Assembler tool (C++)
    - /ildasm – IL Disassembler tool (C++)
    - /bcl – Base Class Library (C#)
  - fx/src – some .NET Framework managed libraries (C#)
  - managedlibraries – additional managed libraries (C#)
  - jscript – JScript compiler (C#)

# External Methods

- methods implemented outside a declaring assembly
- declared
  - without a body
  - using modifier extern (a C# keyword)

```
public extern void ExternalMethod();
```

- target implementation defined by custom attributes
  - DllImportAttribute
    - binding to an unmanaged static entry point of a specified DLL
    - converted to P/Invoke by C# compiler

```
[DllImport("kernel32.dll", EntryPoint="GetDiskFreeSpaceEx")]  
public static extern bool GetDiskFreeSpace (...);
```

- MethodImplAttribute
  - binding to an EE internal method
  - used in BCL (e.g. System.Object.GetHashCode method)

Rotor:  
vm/ecall.cpp

```
[MethodImplAttribute(MethodImplOptions.InternalCall)]  
public extern override int GetHashCode();
```

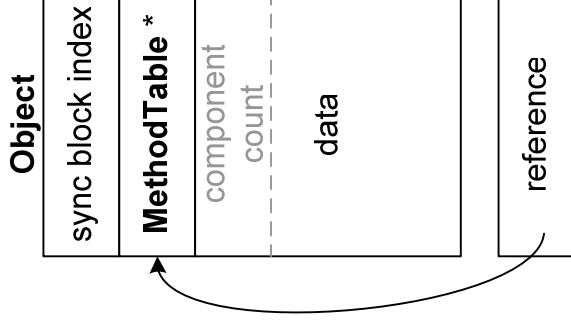
Rotor:

bcl/object.cs

vm/object

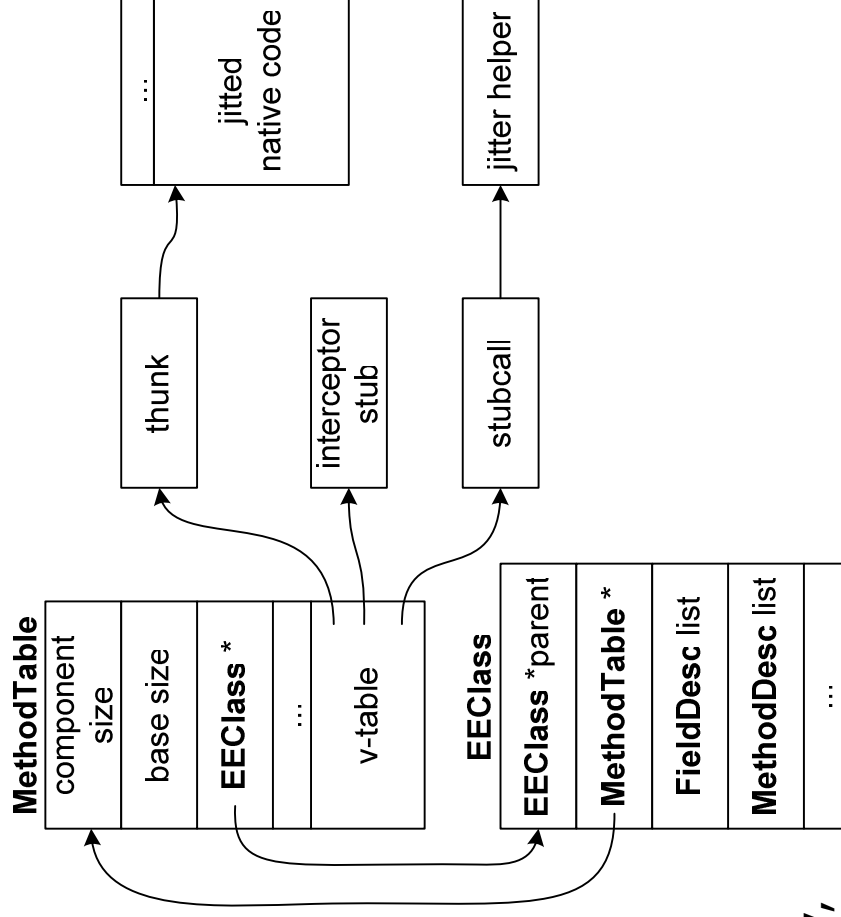
# Managed Object Representation

- two points of view on a managed object
  - a managed world view
    - through class `System.Object` in BCL
  - EE view
    - through class **Object**
- object layout
  - a sync block index
    - an index to EE internal table of sync blocks
    - associates additional data with the object
      - such as a synchronization lock
  - a pointer to a method table
    - an instance of class **MethodTable**
    - one per loaded type
  - component count
    - valid only for arrays and strings
- some bits of sync block and method table pointer exploited for other purposes (e.g. GC)



# Method Table Content

- an object size (for GC)
  - $\text{BaseSize} + \text{ComponentSize} * \text{ComponentCount}$
- a pointer to an **EEClass** instance
  - run-time metadata
  - allocated separately to make working set smaller
- v-table slots
  - including non-virtuals and statics
  - initialized with *stubcalls*
    - invoke the JITter
  - thunk
    - an indirection allowing native code reclamation
  - interceptor stubs
    - ctor, remoting, profiler, security, internal calls, unboxing, arrays,...



# Synchronization via Monitors

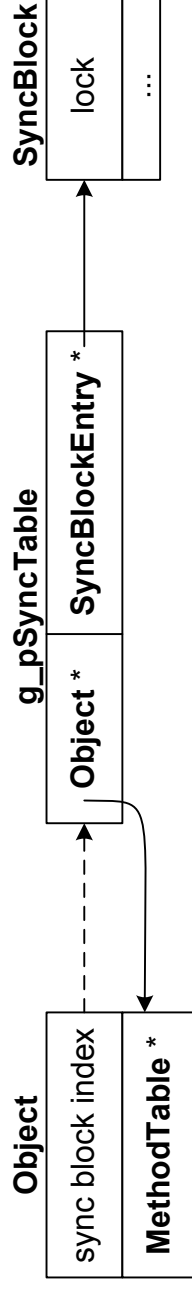
- class System.Threading.Monitor
  - statics: Enter, TryEnter, Exit, Wait, Pulse, PulseAll
  - all take an object treated as a monitor
- C# keyword lock

```
lock(obj) { /* critical section */ }
```

- is equivalent to

```
Monitor.Enter(obj);  
try  
{  
    /* critical section */  
}  
finally  
{  
    Monitor.Exit(obj);  
}
```

- lock is a part of a lazily allocated SyncBlock structure



Rotor:

bcl/string.cs

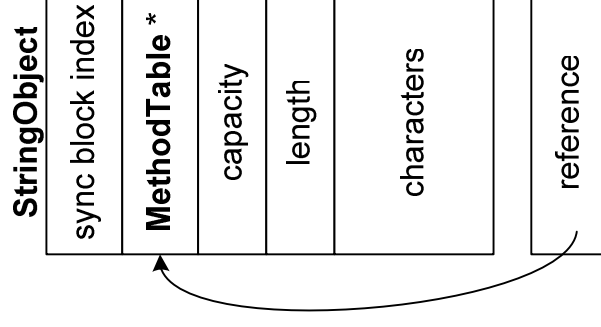
vm/object.h

vm/comstring

vm/stringliteralmap

# Strings

- two points of view on a string object
  - managed: System.String (subclass of System.Object)
  - EE: **StringObject** (subclass of **Object**)
- data internally ends with a `\0` character
  - addresses interoperability with native OS API
  - hidden from outside
  - a string can contain `\0` in the middle
- $\text{length} < \text{capacity} \leq 2 * \text{length}$ 
  - depends on the way how it is built up
- strings are immutable
  - more references can point to a single string object
  - conserves a space
  - time for space trade-off



Rotor:

bcl/string.cs

vm/stringliteralmap

# String Interning

- internal EE hash table
  - holds references to some strings
  - these strings are said to be *interned*
- implicit interning
  - all string literals interned during JIT compilation
- explicit interning
  - string `String.Intern(string)` method
    - adds a string to the internal hash table if not there yet
  - string `String.IsInterned(string)` method
    - returns interned string having the same value as the passed one
    - null if there is no such one

# String Literals

- stored in metadata stream one by one
- loaded by ldstr IL instruction
  - offset in the metadata stream is a part of the instruction
  - loads a reference to a string on evaluation stack
- when a method is being compiled
  - EE internal hash table is looked up for each string literal
  - not found
    - ⇒ a new one is allocated on managed heap
    - ⇒ its value is copied from metadata stream with possible conversion (endianness)
    - ⇒ the literal is interned

bcl/array.cs

vm/object.h

vm/class.h

vm/array

vm/comarrayinfo

vm/comarrayhelpers

# Arrays

- base classes for arrays
  - managed: `System.Array` (subclass of `System.Object`)
  - EE: **ArrayBase** (subclass of **Object**)
- zero-based single-dimensional arrays (vectors)
  - instructions in IL (`newarr`, `ldelem`, `ldlen`, ...)
  - element access range checks
- other arrays (non-vectors)
  - accessed via generated methods
- subclasses of `System.Array` class
  - dynamically generated types (**EEClass** structures)
  - differ in element types and/or ranks
  - generated methods: `Get(...)`, `Set(...)`, `Address(...)`, `.ctor(...)`
    - argument count = rank
    - a call emitted by JIT only if #ranks > 3

# Array Base Layouts

