

# Metadata Internals

**Tomáš Matoušek**  
&  
**Ladislav Prošek**

[tmd.havit.cz/teaching/csharp.htm](http://tmd.havit.cz/teaching/csharp.htm)

# Metadata (recall)

- data describing managed data and code
- associated with all language elements
- stored together with code and data in one file
  
- compilers understand metadata
  - no need for C-like headers in managed languages
  
- programmer can
  - define and use his/her custom metadata
  - query on metadata by reflection API
  
- tools understand metadata as well
  - ILDASM, Visual Studio .NET, ...

# Assemblies

- assembly
  - deployment unit
  - set of one or more files
  - described by manifest (content + additional info)
- files logically contained in an assembly
  - modules (contain IL code)
  - resources
  - other files
- primary module
  - module containing manifest
  - assembly contains at most one primary module
- modules has PE/COFF structure
  - .text section contains IL code, managed resources, metadata

# Metadata Organization

- metadata of each module stored in streams
  - heaps and serialized tables
- heaps
  - contiguous sequences of items
  - types:
    - string heap – zero terminated UTF-8 encoded strings
    - GUID heap – 16B long GUIDs
    - blob heap – binary data, each datum preceded by its length
- tables
  - forms a relational schema
  - entities: types, methods, fields, parameters, ...

# Streams

<b>name</b>	<b>type</b>	<b>content description</b>
#Strings	string heap	names of language elements (types, fields, ...)
#Blob	blob heap	signatures, default values for fields and params, ...
#GUID	GUID heap	GUIDs (each module has a GUID assigned)
#US	blob heap	Unicode string literals (aka user strings)
#~	tables	optimized metadata tables
#-	tables	unoptimized metadata tables

- a persistent module contains the #~ stream (tables are optimized)
- a dynamic module contains the #- stream (tables are not optimized)

# Schema of Metadata Tables

- hardcoded in CLR
- defines columns of each table (type, order and count)
- each table identified by a number (0...63)
- column types:
  - RID<tableId> – an index of a record in explicitly specified table
  - token – a pair [tableId, index]
  - offset in #Strings, #Blob or #GUID stream
  - integer number type
- RIDs used internally
- tokens are parts of instructions and used by metadata APIs
  - e.g. a part of a **call** instruction is token of the method to be called

# Metadata Entity Hierarchies

- parent-children lists
  - a list of all fields in a type
  - a list of all methods in a type
  - a list of all parameters of a method
- optimized tables (#~ stream)
  - children tables are sorted by their parent row index
  - parent entity points to the first child
  - the last child determined by the next parent's first child
- unoptimized tables (#– stream)
  - edit-and-continue scenario
  - intermediate indexing table

# Table Name Suffixes

- -Def tables
  - entities defined in a module containing the table
- -Ref tables
  - entities referred in the module but defined in another one
- -Ptr tables
  - indexing tables (only in #~ stream)

# Metadata Tables

<b>name</b>	<b>contains</b>
Module	module name, GUID
Assembly	information contained in the manifest <ul style="list-style-type: none"><li>• a single record, only in the primary module</li></ul>
AssemblyRef	a list of referenced assemblies (strong names)
ManifestResource	managed resources contained in the module
TypeDef	types defined in the module
FieldDef	fields defined in the module
MethodDef	methods defined in the module
Signature	signatures
CustomAttribute	user defined metadata

... whole schema consists of 44 tables (some are not used in the CLR 1.0) ...

# Adding Custom Metadata

- problem:
  - we want to add a new column to an existing metadata table
  - but tables can't be changed (schema is hardcoded)
- solution:
  - CustomAttribute table contains custom metadata
  - takes advantage of tokens:
    - token can reference items in other tables
    - each row is bound to some table

# CustomAttributes Table

- consists of three columns:
  - parent (token)
    - metadata to which the attribute is assigned
    - refers to a table: Assembly, TypeDef, MethodDef, ...
  - type (token)
    - defines the attribute constructor
    - refers to Method or MemberRef table
  - value (offset in #Blob stream)
    - serialized data
      - constructor's arguments (positional parameters)
      - name-value pairs (named parameters)

# Defining Custom Attribute in C#

```
// custom CLR attribute used on new attribute definition:
[AttributeUsage(AttributeTargets.All, AllowMultiple = true, Inherited = false)]
public class MyAttribute : Attribute
{
    // public constructor (defines attribute's positional parameters):
    public MyAttribute(string s, int i) { this.s = s; this.i = i; }

    // yet another public constructor:
    public MyAttribute(string s) { this.s = s; this.i = 10; }

    // read-write property (defines attribute's named parameters):
    public string X { get { return x; } set { x = value; } }

    // public read-write field defines named parameter:
    public string Y;

    // attribute's private data:
    private string x, y, s;
    private int i;
}
```

- attribute's parameters
  - positional, mandatory: **s**
  - positional, optional: **i**
  - named, optional: **X, Y**

# System.AttributeUsageAttribute

- used on the class defining custom attribute
- three parameters:
  - ValidOn (mandatory, of enumerated type AttributeTargets)
    - language elements on which the attribute can be used
  - AllowMultiple (optional, of type bool)
    - whether more instances of the attribute can be used on one language element
  - Inherited (optional, of type bool)
    - whether the attribute applies also on derived classes and overriding methods

# Pseudocustom Attributes

- not stored in CustomAttribute table
- existing metadata items modified instead
- however, used with the same syntax as CAs
  
- examples
  - DllImportAttribute (on methods)
  - SerializableAttribute (on types)
  - NonSerializedAttribute (on fields)
    - transient keyword in Java

# Inspecting Metadata

- two ways how to inspect metadata programmatically
- via unmanaged APIs
  - low level – working directly with metadata tables
  - used by compilers and tools (ILDASM)
- via System.Reflection classes in BCL
  - higher level of abstraction
    - classes representing types, methods, fields, ...
  - uses runtime metadata representation (recall **EEClass**)
    - some structures created by loader
    - other structures created lazily when requested for the first time
  - no access to IL before FW 2.0

runtime\*.cs  
typehandle.h  
field.h  
method.hpp

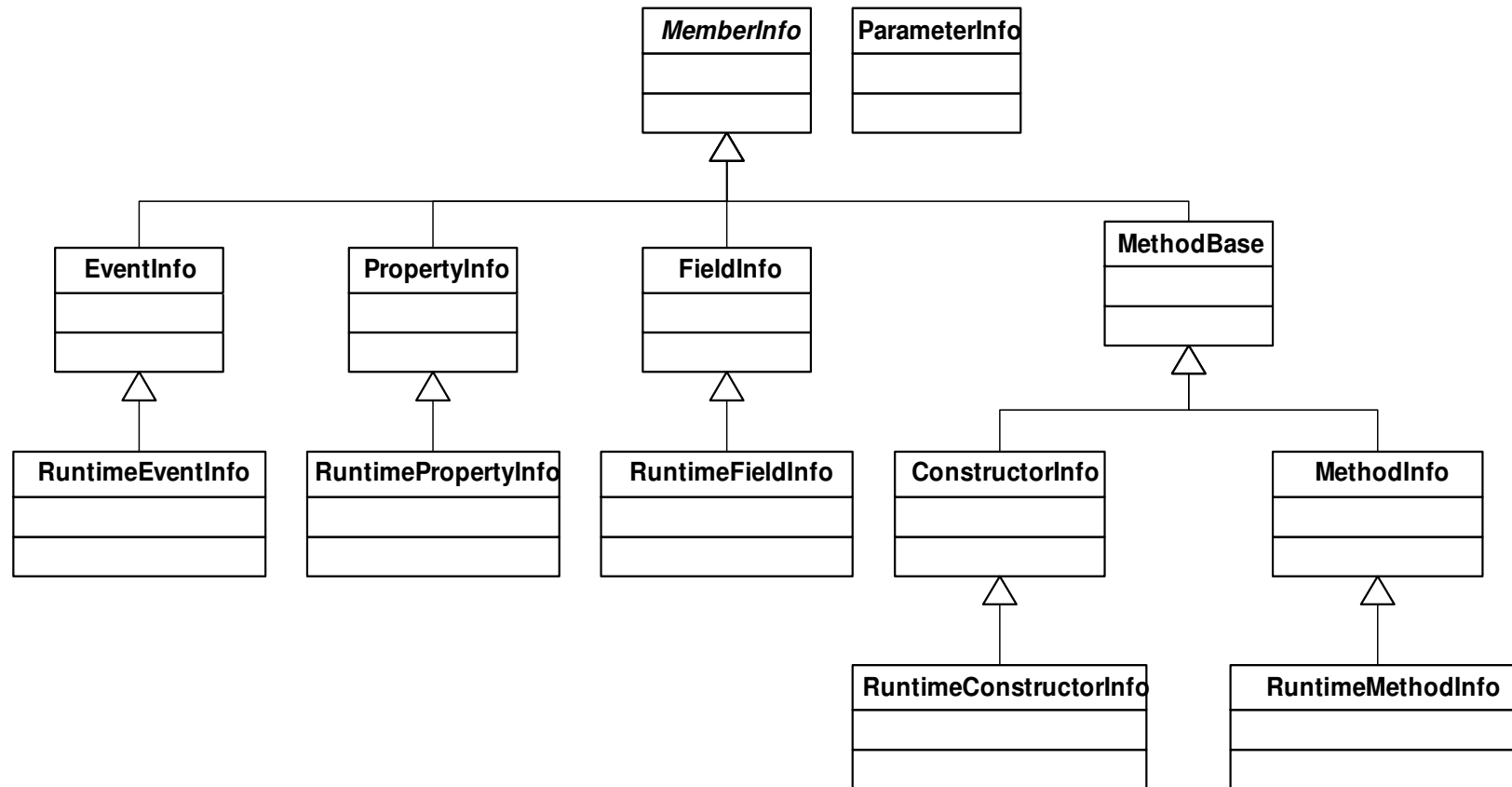
# Runtime Metadata Representation

- structures representing types, fields and methods respectively
  - System.RuntimeTypeHandle (holds a pointer to **TypeHandle**)
  - System.RuntimeFieldHandle (holds a pointer to **FieldDesc**)
  - System.RuntimeMethodHandle (holds a pointer to **MethodDesc**)
- **ldtoken** IL instruction
  - loads a runtime representation of an entity identified by a token
  - implementation of the C# operator typeof(...)

```
ldtoken <type token>
```

```
call class Type Type::GetTypeFromHandle (valuetype RuntimeTypeHandle)
```

# BCL Reflection Classes Hierarchy



- Runtime<member>Info classes has a counterpart class in EE
  - **ReflectBaseObject** class

# Metadata Emission (Reflection.Emit)

- enables program to create new assemblies, types, methods, ...
  - stored in dynamic in-memory structures
  - can be persisted
- builder classes
  - one for each metadata entity
  - represent entities being built
  - inherited from \*Info classes
    - AssemblyBuilder, ModuleBuilder, TypeBuilder, FieldBuilder, MethodBuilder, ParameterBuilder, PropertyBuilder, EventBuilder, ...
- building starts with `AppDomain.CreateDynamicAssembly()`
- IL code emission via `ILGenerator` class
  - instances associated with `MethodBuilders`