

# Reflection.Emit

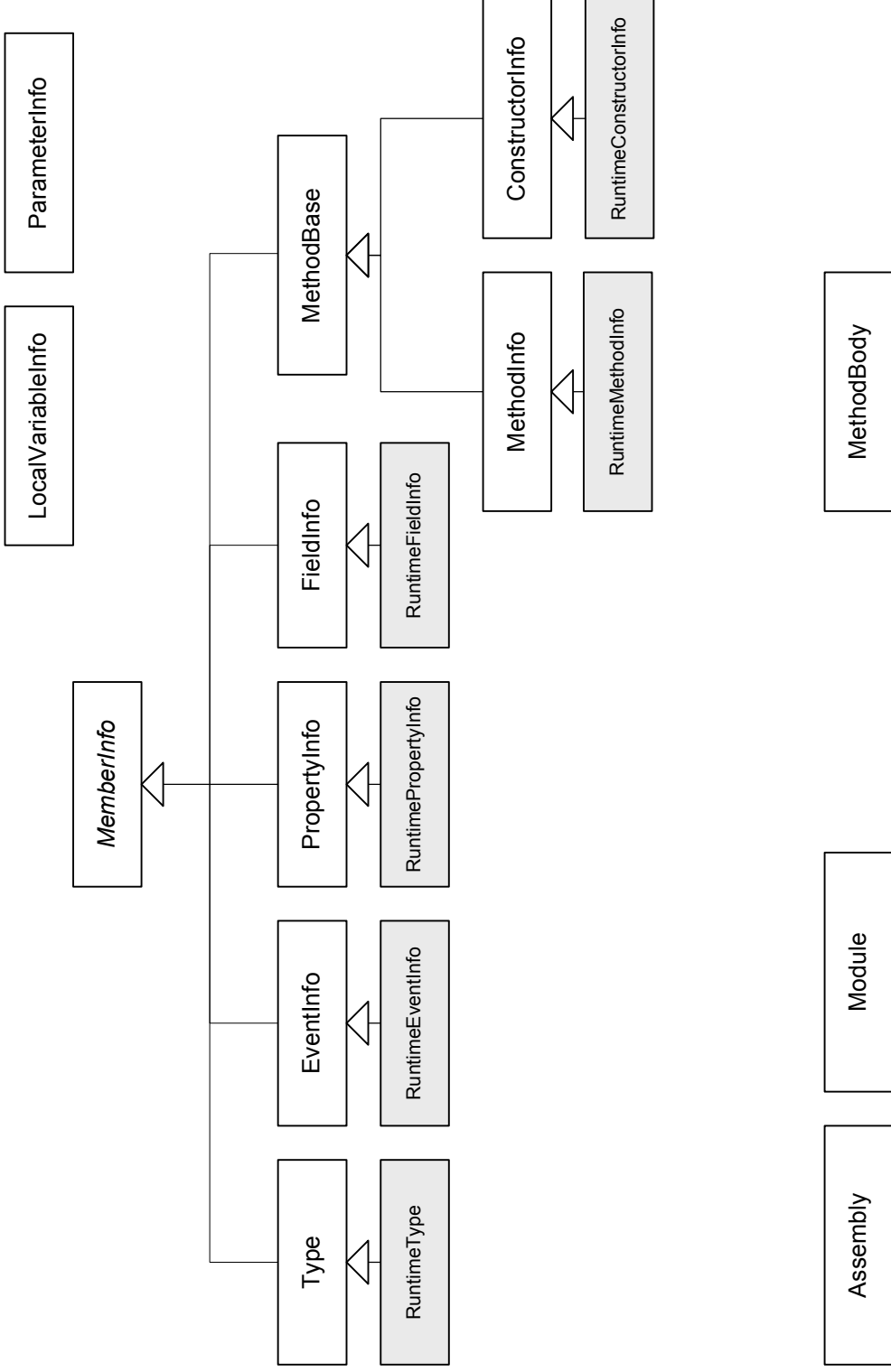
**Tomáš Matoušek**

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

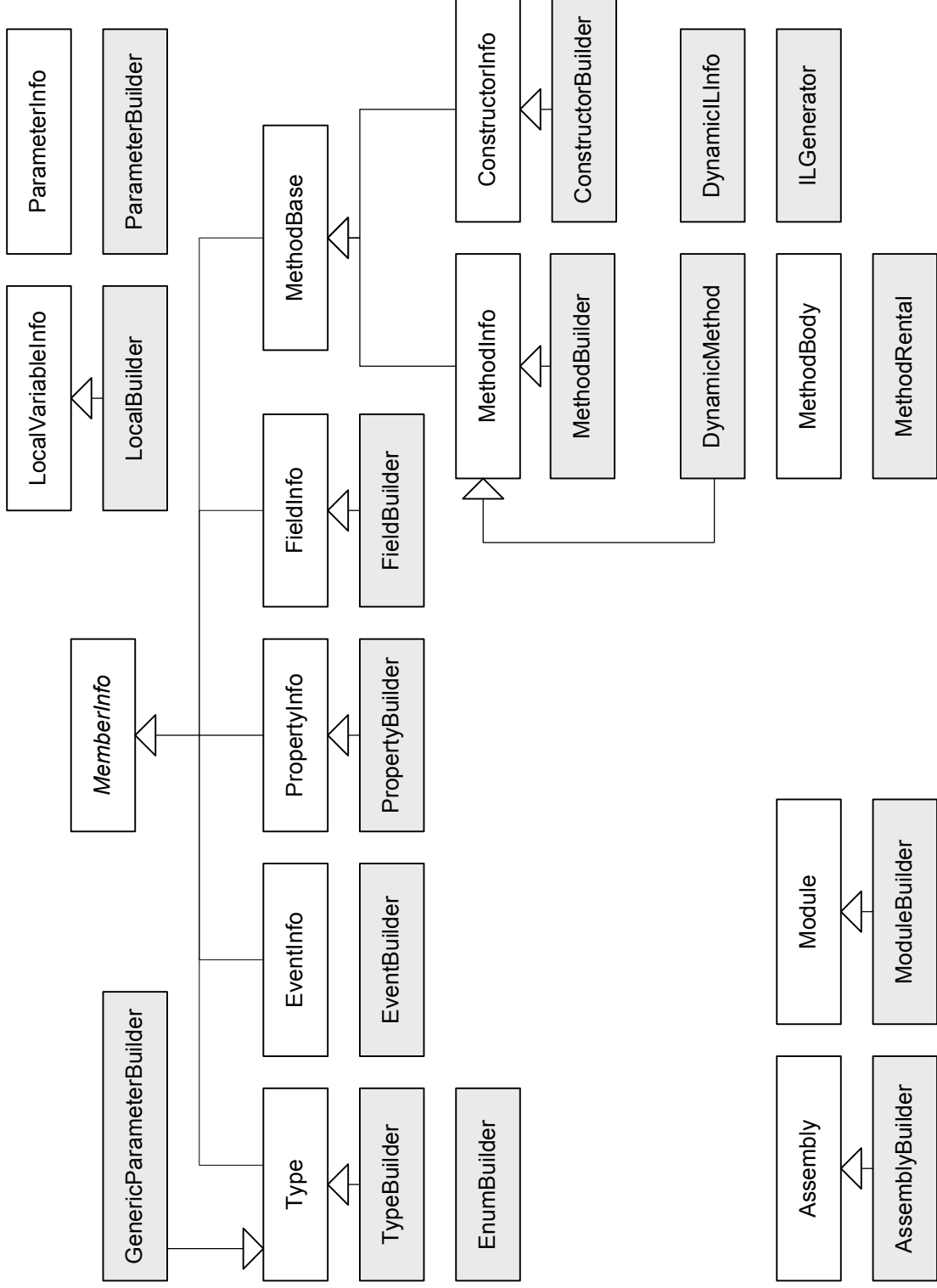
# Metadata Emission

- enables program to create new assemblies, types, methods, ...
  - stored in dynamic in-memory structures
- emitting dynamic assembly
  - AppDomain.DefineDynamicAssembly
  - can be persisted
  - leaks memory (generated metadata and IL not GC'ed)
  - builder classes
    - one for each metadata entity
    - represent entities being built
- Lightweight Code Generation
  - DynamicMethod et al.
  - memory reclaimed
  - some additional useful features

# Reflection Classes



# Reflection.Emit Classes



# Emitting “Hello World” Example

1. `AppDomain.DefineDynamicAssembly`
  - returns `AssemblyBuilder`
2. `AssemblyBuilder.DefineDynamicModule`
  - returns `ModuleBuilder`
3. `ModuleBuilder.DefineType`
  - returns `TypeBuilder`
4. `TypeBuilder.DefineMethod`
  - returns `MethodBuilder`
5. `MethodBuilder.GetILGenerator()`
  - returns `ILGenerator`
6. `ILGenerator.Emit`
  - emits instructions to the internal byte array
7. `TypeBuilder.CreateType`
  - “bakes” the type
  - no changes (except for swapping the method body) possible to the type after this point
8. `AssemblyBuilder.SetEntryPoint`
9. `AssemblyBuilder.Save`
  - persists the assembly

# Generating IL Code

- instance of ILGenerator for each method with body
- instruction emission
  - Emit(OpCodes, param)
- local variables
  - DeclareLocal(Type)
    - returns LocalBuilder instance representing the variable
    - usage
      - Emit(OpCodes.LdLoc, local\_builder);
      - local\_builder.SetLocalSymInfo(variable\_name);
- branches
  - DefineLabel
    - returns the Label structure representing the label
  - MarkLabel
    - associates the label to the current offset in IL stream
  - usage
    - Emit(OpCodes.Br, label);
    - Emit(OpCodes.Switch, new Label[] { case1\_label, case2\_label });

# Code Patterns

for, while, do-while loops

```
<initialization> // for
br l_cond        // for, while

l_body: <body>
l_cont: <increment>
l_cond: <condition>
brtrue l_body
l_break:
```

switch over integers (jump table)

```
<load integer value>
switch (2, l_case0, l_case1)
<case for values above 1>
br l_end
l_case0: <case for value 0>
br l_end
l_case1: <case for value 1>
br l_end
l_end:

Label l_case0 = il.DefineLabel();
Label l_case1 = il.DefineLabel();
Label l_end = il.DefineLabel();

// emit integer value load
il.Emit(OpCodes.Switch, new Label
    { l_case0, l_case1 } );

// emit default case
il.Emit(OpCodes.Br, l_end);
il.MarkLabel(l_case0);
// emit case 0
il.Emit(OpCodes.Br, l_end);
il.MarkLabel(l_case1);
// emit case 1
il.Emit(OpCodes.Br, l_end);
il.MarkLabel(l_end);
```

# Adding Debug Information

- PDB file
  - stores various symbolic information
  - e.g. mapping of IL offsets to positions in a source file, names of local variables, scopes of local variables, ...
- prerequisites
  - `AssemblyBuilder.DefineDynamicModule(_ , _ , true)`
    - enables emission of .pdb file
  - `ModuleBuilder.DefineDocument(source_path, ...)`
    - adds information about the source file
    - returns `ISymbolDocumentWriter`
- sequence points
  - binds following IL instructions to a position in a source file
  - `ILGenerator.MarkSequencePoint(symbol_writer, startLine, startColumn, endLine, endColumn);`
  - sequence points preserved by JIT when debugging

# Sequence Points Example

```
11111111111222222222
0123456789012345678901234567
1 for (int i = 0; i < 10; i++)
2 {
3     Console.WriteLine(i);
4 }

il.MarkSequencePoint(writer, 1, 5, 1, 13);
il.Emit(OpCodes.Ldc_I4_0);
il.Emit(OpCodes.Stloc, local_i);
il.Emit(OpCodes.Br, l_cond);

il.MarkSequencePoint(writer, 3, 2, 3, 22);
il.MarkLabel(l_body);
il.EmitWriteLine(local_i);

il.MarkSequencePoint(writer, 1, 24, 1, 26);
il.Emit(OpCodes.Ldloc, local_i);
il.Emit(OpCodes.Ldc_I4_1);
il.Emit(OpCodes.Add);
il.Emit(OpCodes.Stloc, local_i);
il.MarkLabel(l_cond);

il.MarkSequencePoint(writer, 1, 16, 1, 21);
il.Emit(OpCodes.Ldloc, local_i);
il.Emit(OpCodes.Ldc_I4, 10);
il.Emit(OpCodes.Blt, l_body);

ldc.i4.0
stloc i
br l_cond

l_body:
ldloc i
call void [mscorlib] ...

ldloc i
ldc.i4.1
add
stloc i

l_cond:
ldloc i
ldc.i4 10
blt l_body
```

# Managed Method Body

- method body
  - header
    - tiny (1 byte)
      - <code size><sub>6</sub>01
      - limitations
        - code size  $\leq$  64B, stack depth  $\leq$  8 slots, no locals, no SEH
    - fat (12 bytes)
      - 0011<flags><sub>10</sub>11<max stack depth><sub>16</sub>
      - <code size><sub>32</sub>
      - <local variables signature token><sub>32</sub>
      - flags
        - whether locals must be initialized (0011 000x 1011)
        - whether SEH table follows IL code (0011 0001 x011)
  - IL code
  - SEH table
- see CorHdr.h (IMAGE\_COR\_ILMETHOD)

# MethodRental

- SwapMethodBody static method
  - replaces a body of the specified method with given bytes
  - limitations
    - method must be in a baked type in a dynamic module
  - EE copies the body and updates MethodDesc

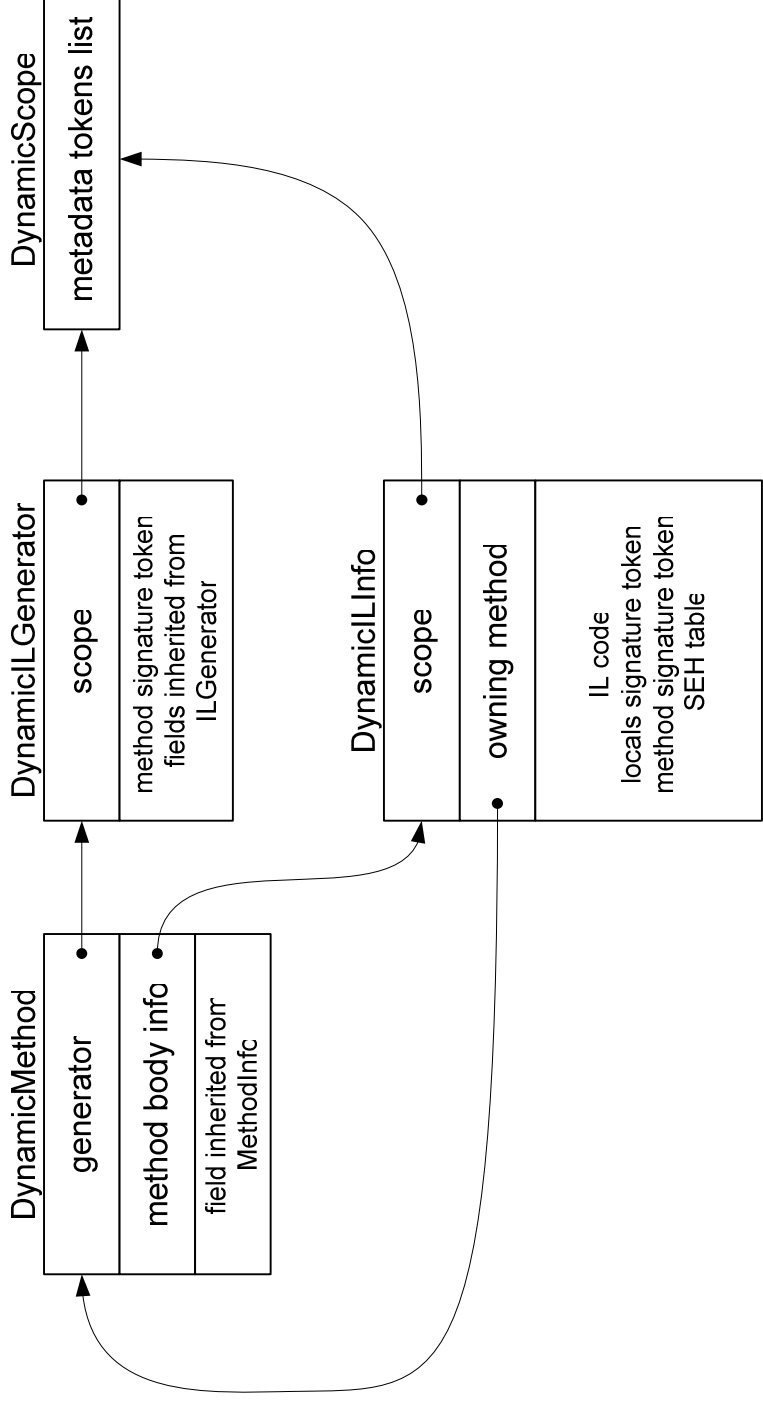
- example (see MSDN)

```
byte[] methodBytes =  
{  
    0x03, 0x30, // flags (0x3003 = no init. locals, no SEH)  
    0x0A, 0x00, // max. stack depth (10 slots)  
    0x02, 0x00, 0x00, 0x00, // code size (2 bytes)  
    0x00, 0x00, 0x00, 0x00, // locals signature token (none)  
    0x17, // 1st instruction (ldc_i4_1)  
    0x2a // 2nd instruction (ret)  
};
```

# Lightweight Code Generation (LCG)

- **DynamicMethod**
  - represents a dynamically emitted method
  - enables to logically add a method to module or type
  - enables to skip some JIT visibility checks
  - construction
    - parameter consistency checks
    - security checks (reflection and member access privileges)
  - IL code emission
    - **GetILGenerator()**
      - returns `DynamicILGenerator` internal class (derived from `ILGenerator`)
    - code is emitted similarly to `ILGenerator`
      - `ldtoken`, `ldftn`, `ldvirtftn` applied on a dynamic method prohibited
  - invocation
    - **CreateDelegate(delegate type, [target instance])**
      - “bakes” the method and creates an instance of the delegate
    - invocations done by calls to the delegate

# LCG Managed Internals



- unlike emission to dynamic assemblies
  - no metadata tables are available for dynamic method
  - IL code needs to use tokens
    - signatures, string literals, constructed types, etc.
  - DynamicScope holds a list of new tokens