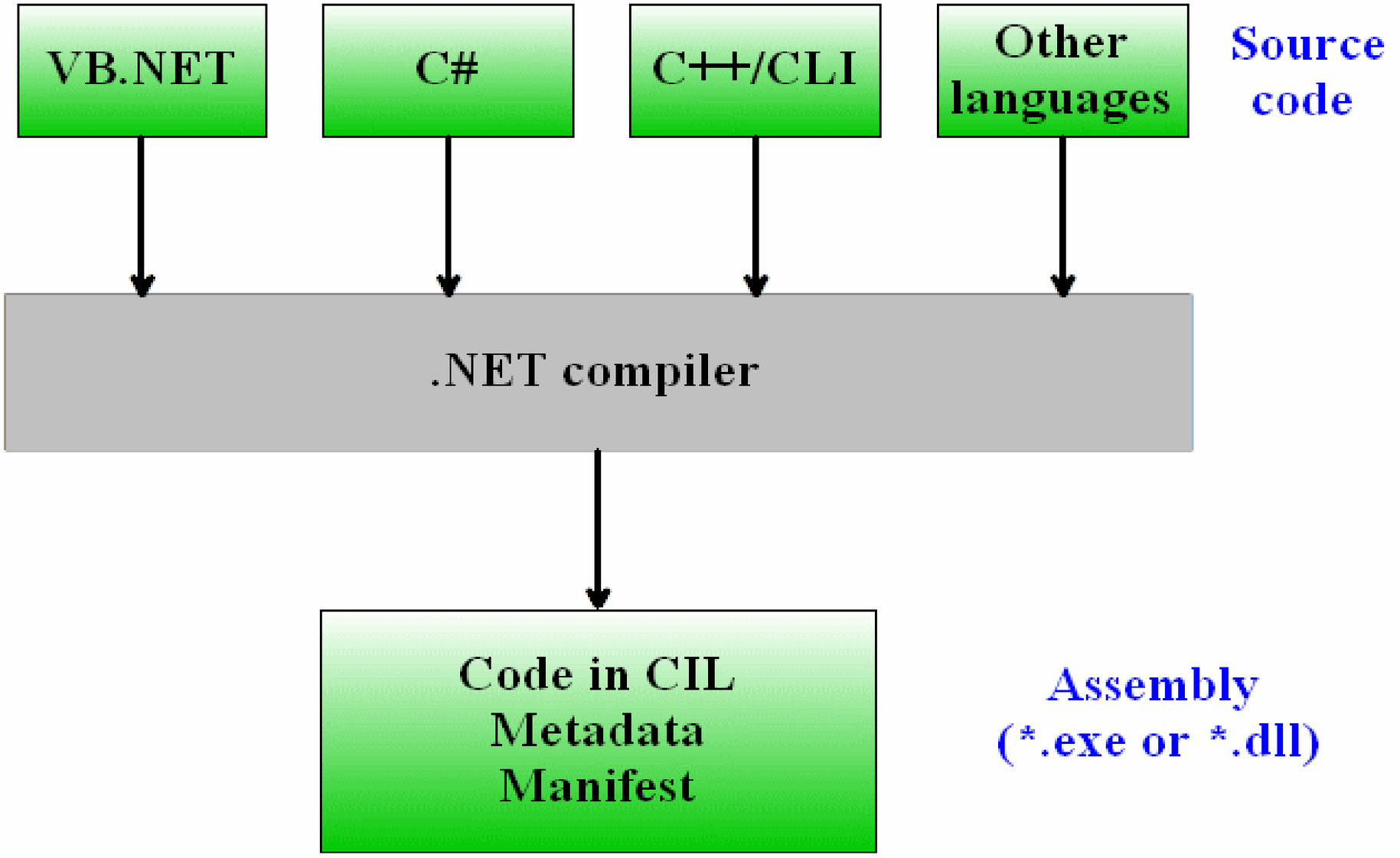
MS .NET Framework

Microsoft .NET Framework is a platform for developing and running software applications. It includes:
•Main concepts.
•Standards.
•Tools for development.
•Tools supporting the software run.
Tools for supporting the software run are integrated into Windows. Tools for development like Visual Studio must be extra downloaded or purchased.

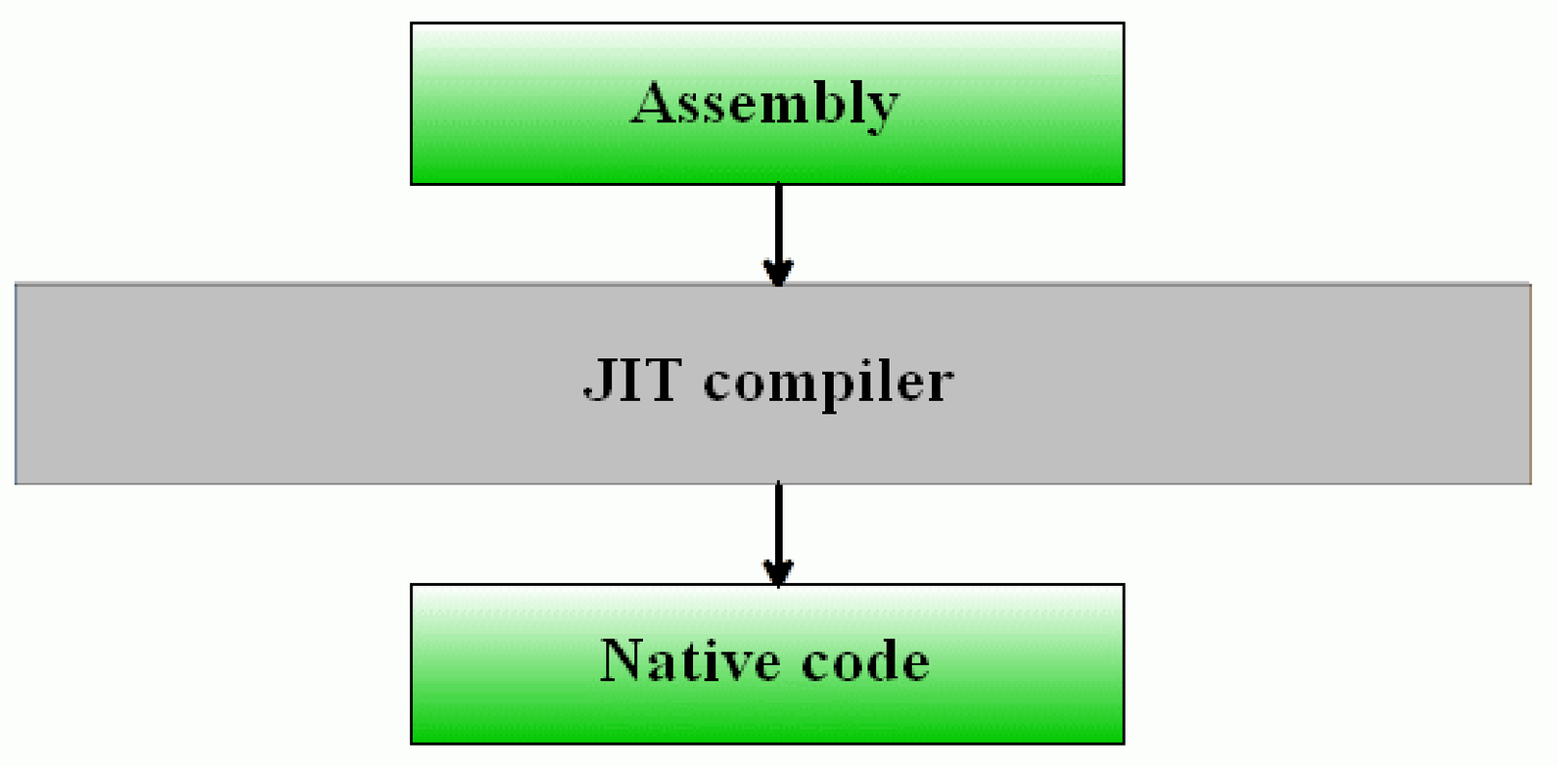
.NET Framework was developed for Windows. Mono is free and open-source project attempting to implement .NET Framework compatible tools for other operating systems like Linux, Mac, Android and Solaris.

.NET Framework 1.0 - 2002 February
.NET Framework 4.7.1 - 2017 October

.NET: development



.NET: running



A .NET program runs in Common Language Runtime (CLR) environment. The Just-In-Time (JIT) compiler converts the CIL code to machine (native) code.
CLR provides also memory managemet, thread management, garbage collection and exception handling.

Assembly

The assembly (*.exe or *dll) is created by Visual Studio compiler.

The Common Intermediate Language (CIL) is conceptually similar to Java bytecode.
The metadata describes every type (classes, structures, etc.) defined in source code,
as well as the members of each type (attributes, functions, etc.).
The assembly manifest describes the assembly itself (assembly name, version number,
list of all files in the assembly, etc.).
To analyze the assembly use utility ILDASM (seldom needed).

If several developers work on one project or the source code files are written in different languages, the compiler must create assemblies with extension *.netmodule. All the netmodules must be together in the same directory. They stay as separate files. The Assembly Linker utility creates an *.exe file containing the multifile assembly manifest.

The multifile assembly may also contain resources like *.jpg or *.bmp pictures.

.NET languages

Some of the .NET languages:
- VB.NET (Visual Basic)
- C# (bases on C++ and Java)
- C++/CLI (extended C++)
- Managed JScript (related to JavaScript)
- J# (similar to Java, discontinued)
- A# (related to Ada)
- F# (functional programming)
- L# (related to Lisp)
- P# (related to Prolog)
- ……

The Common Language Specification (CLS) defines the basic features like data types and programming constructs that .NET languages can (but not must) support. This ensures the complete interoperability among applications created in different languages. Furthermore, it allows to create an application from components written in different languages (for example - one in C# and the other in C++/CLI).

C# code

```
using System;  // in Java: import
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CILExample // in Java: package
{
   class Program
   {
      static void Main(string[] args)
      {
         int a = 1, b = 2;
         int c = a + b;
         Console.WriteLine(c); // in Java: System.out.println
      }
   }
}
```

CIL Code (printed by ILDASM.exe)

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       17 (0x11)
  .maxstack  2
  .locals init ([0] int32 a,
           [1] int32 b,
           [2] int32 c)
  IL_0000:  nop
  IL_0001:  ldc.i4.1
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.2
  IL_0004:  stloc.1
  IL_0005:  ldloc.0
  IL_0006:  ldloc.1
  IL_0007:  add
  IL_0008:  stloc.2
  IL_0009:  ldloc.2
  IL_000a:  call       void [mscorlib]System.Console::WriteLine(int32)
  IL_000f:  nop
  IL_0010:  ret
} // end of method Program::Main
```

A sample from metadata (printed by ILDASM.exe)

```
ScopeName : CILExample.exe
MVID      : {4B672B71-D942-492E-93E8-15AC768C5A17}
Global functions
-------------------------------------------------------
Global fields
-------------------------------------------------------
Global MemberRefs
-------------------------------------------------------
TypeDef #1 (02000002)
-------------------------------------------------------
    TypDefName: CILExample.Program  (02000002)
    Flags     : [NotPublic] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit]  (00100000)
    Extends   : 01000001 [TypeRef] System.Object
    Method #1 (06000001) [ENTRYPOINT]
    -------------------------------------------------------
        MethodName: Main (06000001)
        Flags     : [Private] [Static] [HideBySig] [ReuseSlot]  (00000091)
        RVA       : 0x00002050
        ImplFlags : [IL] [Managed]  (00000000)
        CallCnvntn: [DEFAULT]
        ReturnType: Void
        1 Arguments
            Argument #1:  SZArray String
        1 Parameters
            (1) ParamToken : (08000001) Name : args flags: [none] (00000000)
```

Data types (1)

The data types allowed in .NET languages are

- Classes
- Structures (the keyword in C# is struct)
- Interfaces (conceptually analogous to Java interfaces)
- Enumerations (the keyword in C# is enum, conceptually analogous to Java enumerations)
- Delegates

The members of classes and structures may be

- Fields (i.e. attributes)
- Methods (i.e. functions)
- Properties
- Events

Data types (2)

```
class Rectangle {……..}
Rectangle r1 = new Rectangle(10, 20); // r1 is a variable of reference type
// The rectangle is allocated on the heap and deleted by the garbage collector
// We may say that actually r1 is a pointer
Rectangle r2; // null pointer
r2 = r1; // no copying, simply r1 and r2 point to the same object

struct Point {……..}
Point p1 = new Point(5, 6); // p1 is a variable of value type
// The point is allocated on the stack
Point p2 = new Point(); // default constructor is called, fields initialized to 0
Point p3; // the object exists but the fields are not initialized
string s = p3.ConvertToString(); // error, fields have no values
Point p4 = p1; // object p4 is the copy of p1

int i; // actually a struct of type System.Int32; int is simply the alias of Int32
byte b; // actually a struct of type System.Byte; byte is simply the alias of Byte
i = 10; // handling i as a primitive type
Console.WriteLine(i.ToString()); // handling i as a struct
```

C# class example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace CILExample
{
   class Rectangle
   {
      // Fields
      private int m_width = 1;
      private int m_height = 1;
      // Properties
      public int Width
      {
         get { return m_width; } // accessor function
         set // accessor function
         {
            if (value >= 0) // default argument
               m_width = value;
            else
               throw new ArgumentOutOfRangeException("Illegal width");
         }
      }
```

```
        public int Height
        {
            get { return m_height; }
            set
            {
                if (value >= 0)
                    m_height = value;
                else
                    throw new ArgumentOutOfRangeException("Illegal height");
            }
        }
        // Contructors
        public Rectangle() { }
        public Rectangle(int x, int y)
        {
            Width = x;    // The same as m_width = x, we use the "set" accessor here
                          //  If x < 0, the constructor throws exception
            Height = y;
        }
        // Methods
        public int Area()
        {
            return m_width * m_height;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace CILExample
{
   class Program
   {
      static void Main(string[] args)
      {
         Rectangle r = new Rectangle();  // Allocating an object on the heap
                                         // r is a variable of reference type
         try
         {
            r.Height = 10; // Usage of properties
            r.Width = 20;
         }
         catch (ArgumentOutOfRangeException ex)
         {
            Console.WriteLine(ex.Message);
            return;
         }
         Console.WriteLine(r.Area());
      }
   }
}
```

C# struct example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace CILExample
{
    struct Point
    {
        // Fields (initialization not allowed)
        private int m_x;
        private int m_y;
        // Properties
        public int X {
            get { return m_x; }
            set { m_x = value; }
        }
        public int Y
        {
            get { return m_y; }
            set { m_y = value; }
        }
        // Contructors
        public Point(int x, int y)
        {
            m_x = x;
            m_y = y;
        }
```

```
        // Methods
        public String ConvertToText()
        {
            return "Point X = " + m_x + ", Y = " + m_y;
        }
    }
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace CILExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Point p1 = new Point(5, 6); // Allocating an object on the stack
            Console.WriteLine(p1.ConvertToText());
            Point p2 = new Point(); // Allocating an object on the stack
                        // Each struct has default constructor that initializes all the fields to zero
            p2.X = 10;
            p2.Y = 20;
            Console.WriteLine(p2.ConvertToText());
             return;
        }
    }
}
```

Data types (3)

In C++:

```
class Button
{
    ……..
public:
    void (*m_pAction)(int, int); // pointer to function to be called when
                                 // the button was clicked
    ……
};

void Action (int i, int j) {……..} // to be written by the application
                                  // developer

Button *pb = new Button;
pb->m_pAction = ::Action; // defining a particular button
```

Data types (4)

In Java:

```
JButton m_Button = new JButton();
ActionListener[] listeners = m_Button.getActionListeners();
// "listeners" is the array of pointers to objects of classes implementing interface
// ActionListener. All those classes have the actionPerformed() method. When
// the button has been clicked, the JVM calls all the actionPerformed() methods
// one after another.

m_Button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
            ……….
    }
}); // Create new class implementing interface ActionListener , new object of that
    // class and insert the new object into the array of listeners.
```

Data types (5)

```
void (*pAction)(int, int); // pointer pAction may point to any C++ function that takes 2
                           // integers as parameters and has no return value
public delegate void Action(int i1, int i2); // delegate Action can encapsulate any C#
                                             // method that takes 2 integers as parameters
                                             // and has no return value
```

Delegates are on the same level with classes, structures and enumerations.
In some other class we may declare a field:

```
public class …….. {
……….
public Action m_Action; // pointer to delegate of type Action
……..
public void ProcessAction(int i, int j) {……..}
……..
}

m_Action = new Action(ProcessAction); // create the delegate as object and force to point
                                      // to method ProcessAction
m_Action(10, 20); // invoke the delegate, actually ProcessAction(10, 20) is called
```

Data types (6)

```
public delegate void ClickHandler(int i1, int i2);
public class …….. {
……….
public event ClickHandler m_Click;  // event is always directly tied to a delegate
……….
public void OnClick1(int i, int j) {……}
public void OnClick2(int i, int j) {……}
……….
}

m_Click += new ClickHandler(OnClick1); // associate event with handlers
m_Click += new ClickHandler(OnClick2);

m_Click(); // fires the event, EventHandler 1 and 2 are called
```

C++/CLI (1)

Software developed using the .NET environment: managed code.
Software developed without .NET environment: unmanaged (or native) code.
Interop: managed code functions may call functions from unmanaged DLL-s.
Class containing methods that need interop must include descriptions (i.e. prototypes) of the needed unmanaged functions.
Example:

```
class ClassForSomething
{
  [DllImport("setupapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
  public static extern IntPtr SetupDiGetClassDevs(ref Guid classGuid,
                                    IntPtr enumerator, IntPtr hwndParent, UInt32 flags);
   ……..……..
   private int DoSomething()
   {
     IntPtr HWDeviceInfo = SetupDiGetClassDevs(ref guid, IntPtr.Zero,
                                                          IntPtr.Zero, 0x00000002);
     ……..……..
    }
}
```

C++/CLI (2)

Goal: insert existing sophisticated (third-party) C++ classes into a new .NET application.
Problems:
.NET and C# does not support C/C++ style pointers and pointer arithmetics.
C/C++ does not support reference and value types, delegates, events, etc.
Recode from C/C++ to C# is too time-consuming and expensive.
Solution: write the new .NET application or just some of its modules not in C# but in C++/CLI.
A C++/CLI project may include C++ classis written in standard C++ without any modifications on them (so called native classes).
Also, C++/CLI project must include classes following the rules unknown in standard C++ version 11 (so called managed classes).

```
class Cccc {……};  // native class
struct Ssss {……};  // actually also native class
ref class Cccc {……..};  // managed class, compatible with .NET classes
ref struct Ssss {……..}; // actually also managed class
value class Cccc {……..};  // managed class, compatible with .NET structs
value struct Ssss {……..}; // actually also managed class
```

C++/CLI (3)

```
class C_native {…….};
C_native *p = new C_native(…); // pointer, memory from  native heap
p->fun(…);
delete p;

ref class C_managed {……};
C_managed ^h = gcnew C_managed(….);
                        // tracking handler, memory from managed heap
h->fun(….);
```

Tracking handlers behave as reference type values in C#. For example:

```
C_managed ^h1; // h1 value is nullptr (not 0)
h1 = h; // no copying, h1 and h are pointing to the same object
h1 = h + 1 // error, no tracking handler arithmetics
delete h; // error, memory in managed heap is released by garbage collector
```

C++/CLI (4)

Arrays:

```
int *p = new int[100]; // native
array<int> ^h = gcnew array<int>(100);
                // managed, array is an object of class System::Array
for (int i = 0;  i < h->Length; i++)
    h[i] = 100;
```

Strings:

```
wchar_t *p = L"Hello"; // native
String ^h = L"Hello"; // managed, string is an object of class System::String
String ^h1 = gcnew String(L'a', 5);
String ^h2 = String::Copy(h1);
Console::WriteLine(h1 + h2); // get Helloaaaaa
```

C++/CLI supports properties, delegates, etc.