# High-Level Language Virtual Machine Architecture

# Contents

# Java Instruction Set

- Instruction set format
  - opcode byte + zero or more operands
    - Operand can be an index, immediate data, or PC-relative offset
    - Wide & escape code can be used to extend
  - Each primitive type has its own instruction set
    - E.g., iadd, fadd, …
    - Operand types must match the opcode
    - So, bytecode ISA is called a typed stack machine

# Data-Movement Instructions

- Pushing constants onto the stack
  - `aconst_null`, `iconst_1`, `ldc` (via constant pool), `bipush` (direct)
  - There can be different instructions for the same function
- Stack manipulation instructions
  - `pop, dup, swap`
- Move values bet' operand stack and locals in the stack frame
  - `iload_1, aload 5, istore_2, astore 5`
- Object creation, field access, and check instructions
  - new creates a new object and save the reference on the operand stack
  - `getfield/putfield` move data bet'n object and stack
  - `Getstatic/putstatic` move data bet'n object and method area
  - `checkcast` checks if the top object is an instance of a type

# Conversion and Functional Instructions

- Type Conversion
  - I2f converts the top element from int to float
- Functional Instructions
  - Arithmetic, logical, and shift instructions
  - Only operates on int, float, double, long

# Control Flow Instructions

- Branches and jumps
  - `ifeq, if_icmpeq, lookupswitch`
  - PC-relative branch to a constant offset (no indirection)
- Method call
  - `invoke(virtual|static|special|interface)` <index>
    - Indexes CP where information on method address, arguments, # locals, stack depth can be found
      - ✓ Argument check, frame allocation, push arguments as locals
      - ✓ Jump to the method
      - ✓ Return PC is saved on the stack (in frame data area), but can not be accessed directly (only through return)
  - `ireturn` makes a return after popping an integer and then push it on the stack after removing the frame
- All control paths can be easily tracked
  - No code discovery problem

# Operand Stack Tracking

- For any point in the program, the operand stack state must be the same regardless of the path to the point
- Operand stack tracking at loading for validity check
  - Since control flows can be determined in loading time
  - Loader can also checks for the followings
    - Stack limits
    - Types of arguments to JVM instructions
    - Accesses or assignments to local variables
- Invalid program found via operand stack tracking
  - ex)

```
    iconst_4
    istore_1
Loop:
    aconst_null
    iinc 1 –1
    iload_1
    ifeq Loop
```

➔ Operand stack is not equivalent at Loop.

# Exceptions and Errors

- Exception handling in Java
  - By providing `try`, `catch`, and `finally` blocks

```
try {
  ………..
 }
 catch (exception_type e) {
    ……
 }
 finally {
   …………
}
```

- All exceptions must be handled somewhere
  - If there is no catch block in a excepting method, stack frame is popped until the exception handler is found
- An exception is thrown via `athrow` instruction

# Exception Table

- Use exception table to specify an exception handler

| From | To | Target | Type |
| --- | --- | --- | --- |
| 8 | 12 | 96 | Arithmetic Exception |

It means that if an arithmetic exception is thrown between bytecode 8 and 12, jump to bytecode 96

# Java VM Architecture

## - Exceptions and Errors

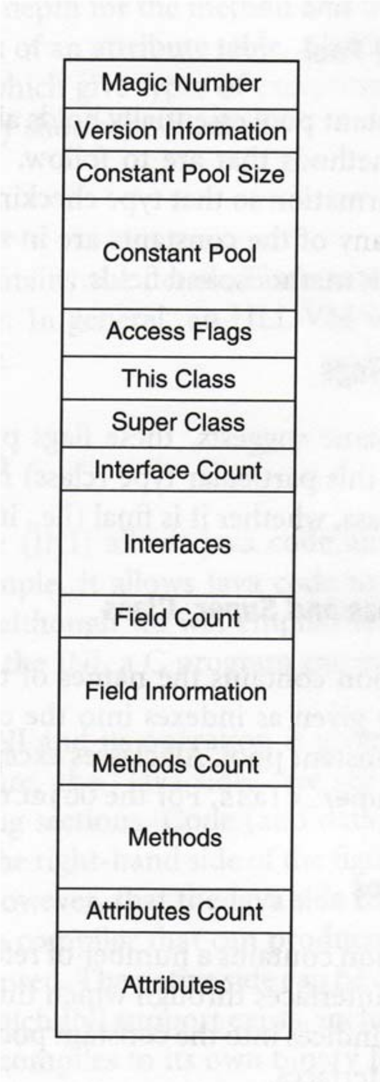
- Exception handler example
  - Example Java code

```
public class ExceptionTest {

    public static void main(String args[]) {

        try {
            java.io.FileInputStream x
            = new java.io.FileInputStream(“myfile”);
        } catch(java.io.FileNotFoundException e) {
            System.out.println(“Not found”);
        } finally {
            System.out.println(“This must be executed”);
        }
    }
}
```

```
{
public ExceptionTest();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return
  LineNumberTable:
   line 1: 0

public static void main(java.lang.String[]);
  Code:
   Stack=3, Locals=3, Args_size=1
   0:   new     #2; //class java/io/FileInputStream
   3:   dup
   4:   ldc     #3; //String myfile
   6:   invokespecial   #4; //Method java/io/FileInputStream."<init>":(Ljava/lang/String;)V
   9:   astore_1
   10:  getstatic       #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   13:  ldc     #6; //String This must be executed
   15:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   18:  goto    52
   21:  astore_1
   22:  getstatic       #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   25:  ldc     #9; //String Not found
   27:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   30:  getstatic       #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   33:  ldc     #6; //String This must be executed
   35:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   38:  goto    52
   41:  astore_2
   42:  getstatic       #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   45:  ldc     #6; //String This must be executed
   47:  invokevirtual   #7; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   50:  aload_2
   51:  athrow
   52:  return
  Exception table:
   from   to  target type
     0    10    21   Class java/io/FileNotFoundException

     0    10    41   any
    21    30    41   any
    41    42    41   any
  LineNumberTable:
   line 6: 0
   line 10: 10
   line 11: 18
   line 7: 21
   line 8: 22
   line 10: 30
   line 11: 38
   line 10: 41
   line 12: 52

}
```

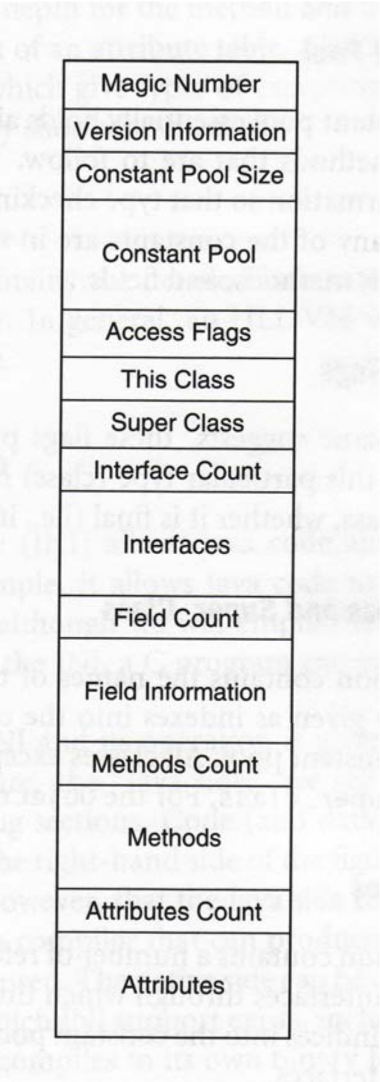
# Structure of Class File

- Binary class file includes metadata + code
- Magic number
  - 0xCAFEBABE(in big-endian order)
- Constant pool (CP)
  - References and constants used by methods
- Access flags
  - Class or interface, public or private, final or not
- This class and super class
  - Names which are given as indexes to CP
- Interfaces
  - Name of interfaces that this class implements



# Java VM Architecture
## - Class File Structure

- Fields
  - Specification of fields declared in this class
- Methods
  - Bytecode instruction stream
- Attributes
  - More details



# Java VM Architecture
## - Class File Structure

ClassStruct_java.txt    ClassStruct.txt

```
000000  ca fe ba be 00 00 00 31 00 20 0a 00 06 00 11 09   Êþº¾...1........
000010  00 12 00 13 08 00 14 0a 00 15 00 16 07 00 17 07   ................
000020  00 18 07 00 19 01 00 01 61 01 00 01 49 01 00 06   ........a...I...
000030  3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43   <init>...()V...C
000040  6f 64 65 01 00 0f 4c 69 6e 65 4e 75 6d 62 65 72   ode...LineNumber
000050  54 61 62 6c 65 01 00 04 74 65 73 74 01 00 0a 53   Table...test...S
000060  6f 75 72 63 65 46 69 6c 65 01 00 10 43 6c 61 73   ourceFile...Clas
000070  73 53 74 72 75 63 74 2e 6a 61 76 61 0c 00 0a 00   sStruct.java....
000080  0b 07 00 1a 0c 00 1b 00 1c 01 00 0b 54 65 73 74   ............Test
000090  20 53 74 72 69 6e 67 07 00 1d 0c 00 1e 00 1f 01    String.........
0000a0  00 0b 43 6c 61 73 73 53 74 72 75 63 74 01 00 10   ..ClassStruct...
0000b0  6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74   java/lang/Object
0000c0  01 00 14 6a 61 76 61 2f 69 6f 2f 53 65 72 69 61   ...java/io/Seria
0000d0  6c 69 7a 61 62 6c 65 01 00 10 6a 61 76 61 2f 6c   lizable...java/l
0000e0  61 6e 67 2f 53 79 73 74 65 6d 01 00 03 6f 75 74   ang/System...out
0000f0  01 00 15 4c 6a 61 76 61 2f 69 6f 2f 50 72 69 6e   ...Ljava/io/Prin
000100  74 53 74 72 65 61 6d 3b 01 00 13 6a 61 76 61 2f   tStream;...java/
000110  69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 01 00   io/PrintStream..
000120  07 70 72 69 6e 74 6c 6e 01 00 15 28 4c 6a 61 76   .println...(Ljav
000130  61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56   a/lang/String;)V
000140  00 21 00 05 00 06 00 01 00 07 00 01 00 09 00 08   .!..............
000150  00 09 00 00 00 02 00 01 00 0a 00 0b 00 01 00 0c   ................
000160  00 00 00 1d 00 01 00 01 00 00 00 05 2a b7 00 01   ............*...
000170  b1 00 00 00 01 00 0d 00 00 00 06 00 01 00 00 00   ±...............
000180  01 00 01 00 0e 00 0b 00 01 00 0c 00 00 00 25 00   ..............%.
000190  02 00 01 00 00 00 09 b2 00 02 12 03 b6 00 04 b1   .......²....¶..±
0001a0  00 00 00 01 00 0d 00 00 00 0a 00 02 00 00 00 06   ................
0001b0  00 08 00 07 00 01 00 0f 00 00 00 02 00 10         ..............
```

# Native Method Support

- Java Native Interface (JNI)
  - Allows Java code and native code to interoperate
    - E.g., Java code call a routine compiled from C

# Java APIs

- Java provides abundant APIs
  - Network computing, component-based S/W, GUIs
  - Each edition provides different API packages
- J2SE (Standard Edition)
  - API for PC users and client-side applications, JavaBeans
- J2EE (Enterprise Edition)
  - API for developing large enterprise software infrastructure
  - EJB, servlet, JSP, JMS,
- J2ME (Micro Edition)
  - Light-weight platform for embedded system

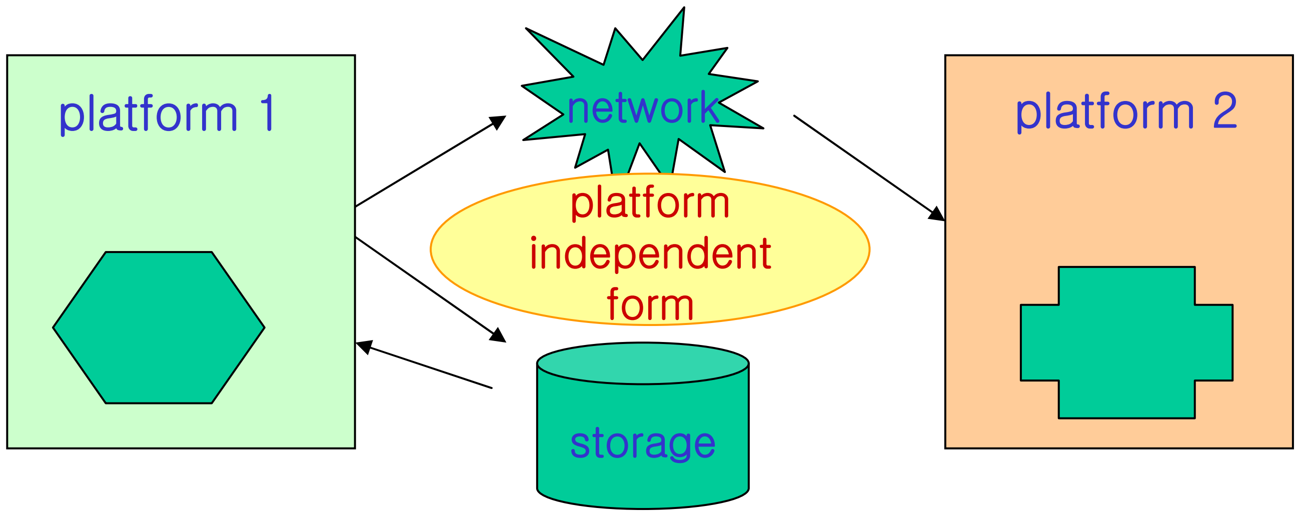
# Java Core APIs

- *java.lang*
  - Core Java programming language classes
  - `Object` class: superclass of all Java classes
  - `Class` class: each loaded class has a `Class` object for it; it allows to extract information on the class using *refelction*
  - `Thread` class, `SecurityManager` class, …
- java.util
  - Fundamental data structures
  - `Vector` class, `Enumeration` interface, `Hashtable` class, …
- *java.awt*, *java.io*, *java.net*

# Serialization and Reflection

Allows exposing the feature of an object to outside

- Remote method invocation from one Java program to another, on different platforms, with object arguments
  - Argument or return value object must be converted to an implementation-independent form due to platform difference
- Objects created by a program may persist bet'n runs, stored on a disk or a flash memory (e.g., Java card)
  - Convert the object into an implementation-independent form



# Serialization and Reflection

- Serialization: the process of converting an object into an implementation-independent form
  - Object must be declared to implement `Serialzable` interface
  - Other objects referenced in the object must be serialized
  - Requires ***reflection***, look inside an object to find all members
- Other reflection usages
  - When a running program is given a reference to unknown object
  - Component-based programming (JavaBeans) often requires a graphical tool which must read bean’s design patterns
  - *Java.lang.reflect* API is an interface to a class’ `Class` object
    - Which includes description of a class

# Threads

- Thread
  - Multithreading support is provided by java.lang.Thread class (and Runnable interface)
  - Thread execute run() method during its lifetime.
- Synchronization through monitor
  - Required when we have a synchronized method
  - Supported by monitorenter and monitorexit bytecode
  - Locks are associated with each synchronized object
  - Other synchronization support: Notify(), notifyAll(), wait()

# Java APIs

- Synchronization Example

```
public class SyncTest extends Thread {
        static int flag = 0;

        public void run() {
                synchronized(SyncTest.class) {
                        if(flag == 1) {
                                SyncTest.class.notifyAll();
                        }
                        else {
                                flag = 1;
                                try {
                                        SyncTest.class.wait();
                                } catch(Exception e) {
                                }
                        }
                }
        }

        public static void main(String args[]) {
                SyncTest thread1 = new SyncTest();
                SyncTest thread2 = new SyncTest();
                thread1.start();
                thread2.start();
        }
}
```

```
public void run();
  Code:
   Stack=2, Locals=4, Args_size=1
   0:   getstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   3:   ifnonnull       18
   6:   ldc     #8; //String SyncTest
   8:   invokestatic    #9; //Method class$:(Ljava/lang/String;)Ljava/lang/Class;
   11:  dup
   12:  putstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   15:  goto    21
   18:  getstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   21:  dup
   22:  astore_1
   23:  monitorenter
   24:  getstatic       #10; //Field flag:I
   27:  iconst_1
   28:  if_icmpne       58
   31:  getstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   34:  ifnonnull       49
   37:  ldc     #8; //String SyncTest
   39:  invokestatic    #9; //Method class$:(Ljava/lang/String;)Ljava/lang/Class;
   42:  dup
   43:  putstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   46:  goto    52
   49:  getstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   52:  invokevirtual   #11; //Method java/lang/Object.notifyAll:()V
   55:  goto    90
   58:  iconst_1
   59:  putstatic       #10; //Field flag:I
   62:  getstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   65:  ifnonnull       80
   68:  ldc     #8; //String SyncTest
   70:  invokestatic    #9; //Method class$:(Ljava/lang/String;)Ljava/lang/Class;
   73:  dup
   74:  putstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   77:  goto    83
   80:  getstatic       #7; //Field class$SyncTest:Ljava/lang/Class;
   83:  invokevirtual   #12; //Method java/lang/Object.wait:()V
   86:  goto    90
   89:  astore_2
   90:  aload_1
   91:  monitorexit
   92:  goto    100
   95:  astore_3
   96:  aload_1
   97:  monitorexit
   98:  aload_3
   99:  athrow
   100: return
  Exception table:
   from   to  target type
    62    86    89   Class java/lang/Exception

    24    92    95   any
    95    98    95   any
```

# Common Language Infrastructure (CLI)

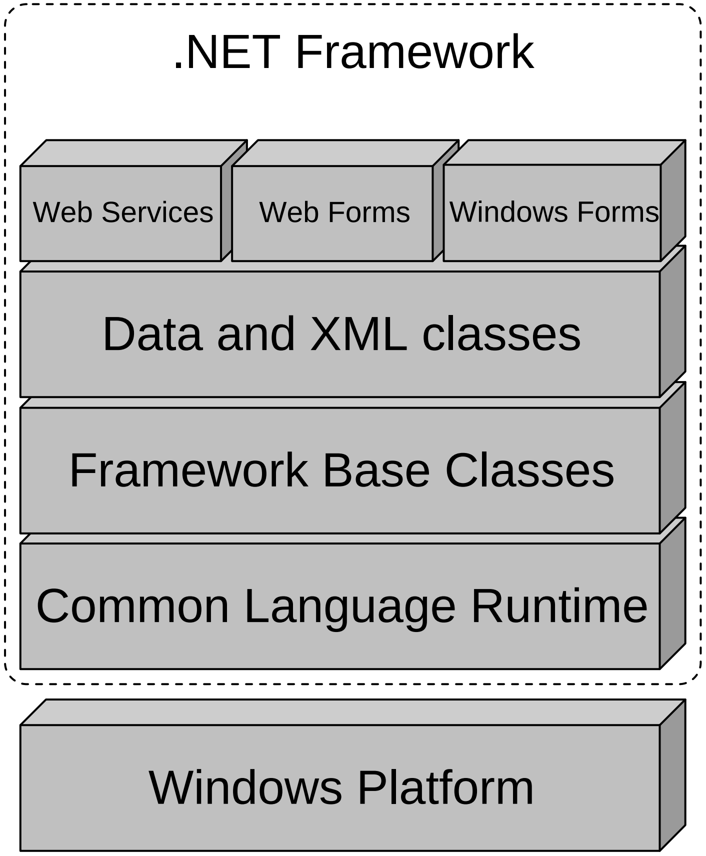
- A VM architecture of Microsoft .NET framework
  - Common language runtime (CLR) is MS implementation of CLI
  - Other implementations: DotGNU Portable .NET, MONO
- Terminology comparison to Java
  - JVM Architecture ←→ CLI
    - Analogous to an ISA
  - JVM Implementation ←→ CLR
    - Analogous to an ISA implementation
  - Java bytecode ←→ CIL (common intermediate lang)
    MSIL (MS intermediate lang.)
    - The instruction part of the ISA
  - Java Platform ←→ .NET framework
    - ISA implementation + libraries

# Microsoft .NET Overview

- A software component that can be added to Windows
  - Is included in Vista
  - Intended to be used by most new applications for Windows
- Provides large APIs and manages program execution
  - User interface, DB connectivity, cryptography, web application development, numeric algorithms, and network
  - Programs for .NET are executed in a managed environment
    - CLR who manages portability, security, memory, and exception

# Design Goals of .NET

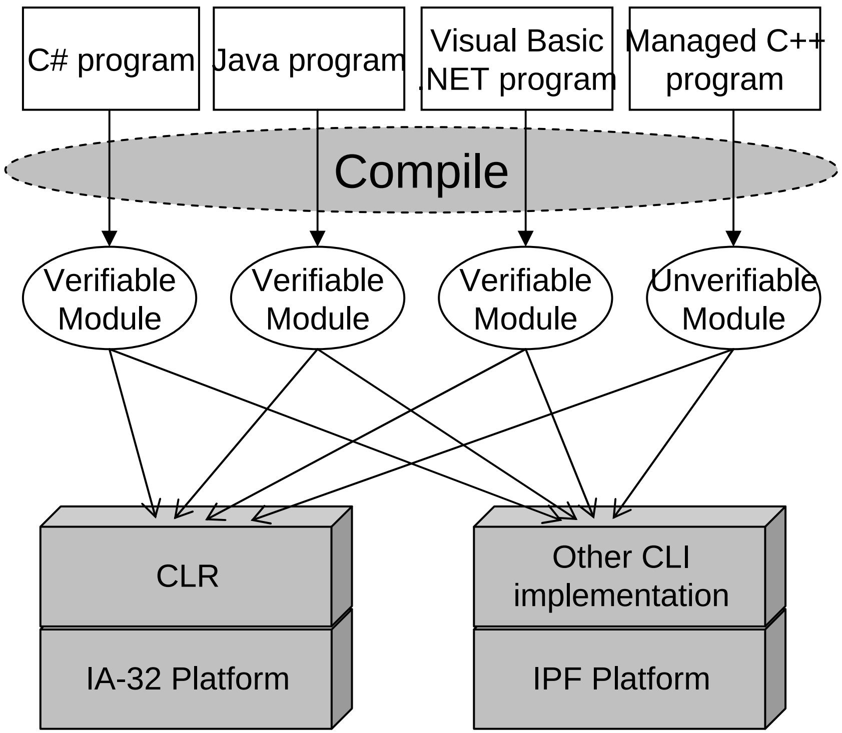
- Interoperability
  - Allows interaction between new and older applications
- Component infrastructure
  - "plug-and-play"
- Language integration
  - COM was binary reuse, not language integration
- Others
  - Reliability, Security, Internet interoperation,Simple deployment



# The Common Language Infrastructure

- Similarities with Java environment
  - Support OOP within a managed runtime environment,
  - Includes built-in protection and garbage collection
  - Platform independent
- Differences from Java environment
  - Not only platform independent, but also HLL independence
    - Support multiple, interoperating HLLs
  - Support both verifiable and unverifiable code (even mixed)
    - Verifiable: C#, Visual Basic .NET, Java, some managed C++
    - Unverifiable: some managed C++, legacy C and C++
    - Invalid programs cannot run in CLI
  - Useful while unsafe legacy code still exists

# Independence and Interoperability

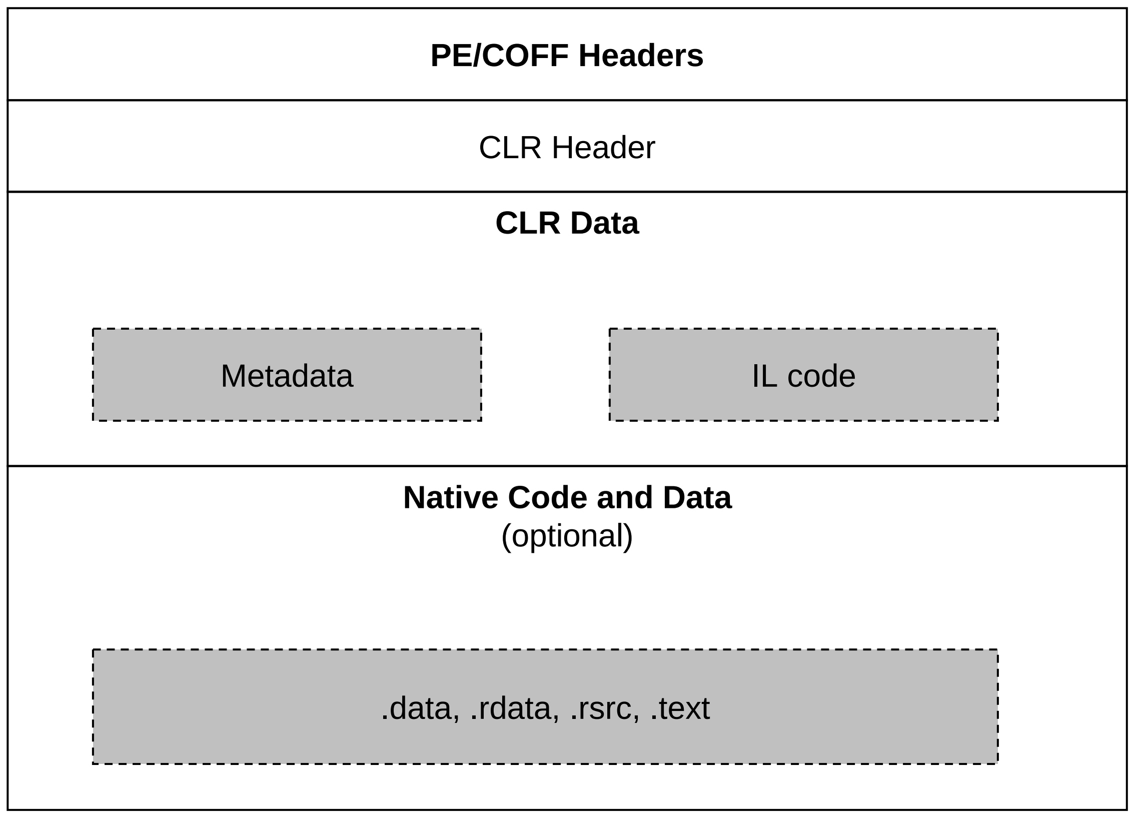



To achieve both,

- Standard rule among language
- Platform independent and verifiable IR
- Platform-independent module specification
- Methodology or specification for plumbing type of objects
- Specification to represent information of program
- Specification of VM runtime
- ……

So CLI, MSIL, PE/COFF, CLS, CTS, etc,. are defined by Microsoft.

# CLI Module Structure



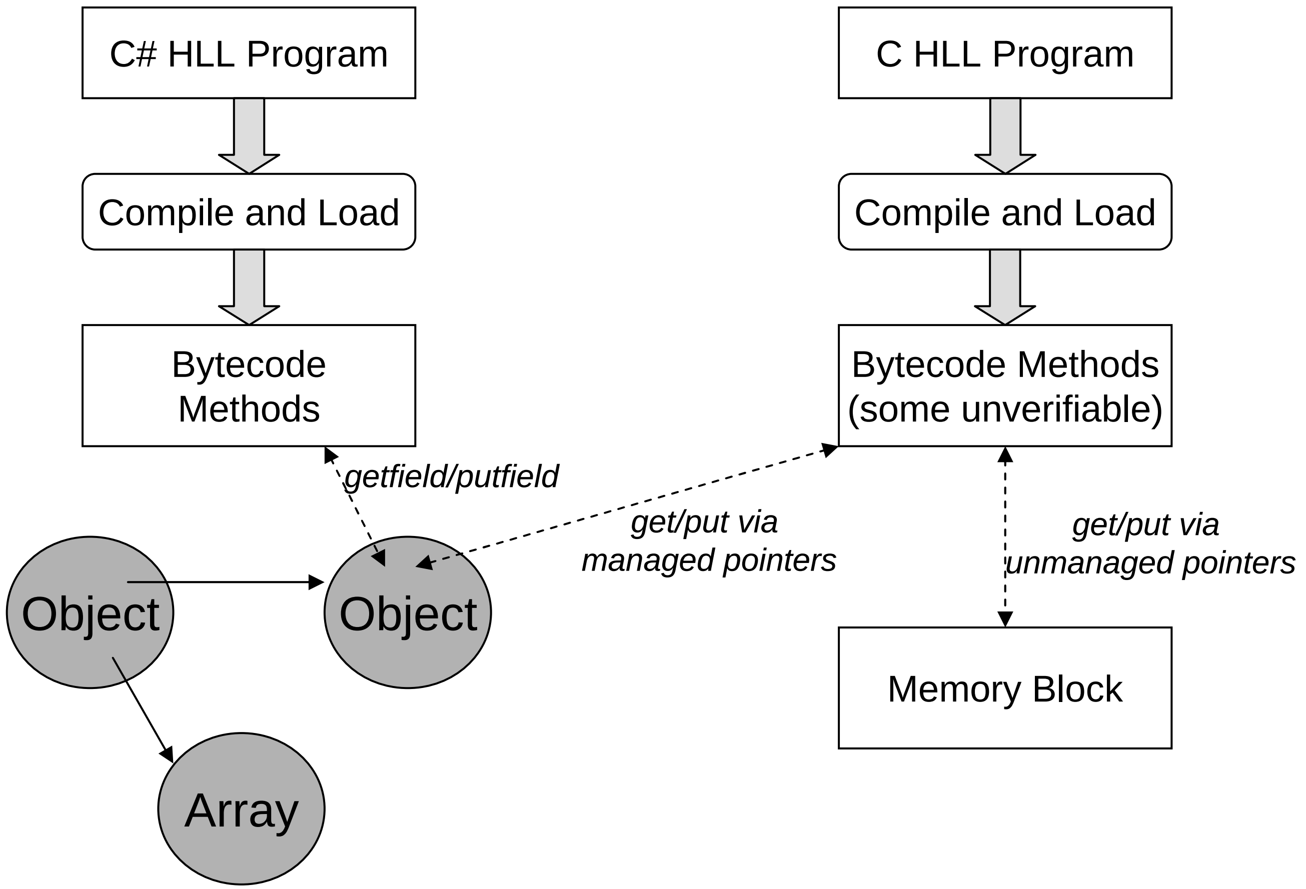
# Microsoft PE/COFF

- Portable Executable and Common Object File Format
  - For interoperability
  - Just like any other executable files to the OS
- Metadata with object definitions and constants
- MSIL bytecode (even for C)
- Native Code and data (e.g., for encryption)

# Interoperability

- Not method level such as JNI
- More integrated way that also extends to data
  - A type defined in a language can be used across languages
- Require change to language implementation
  - Standardization of language → CLS
  - Standardization of type → CTS

# Interoperate example



# Attributes

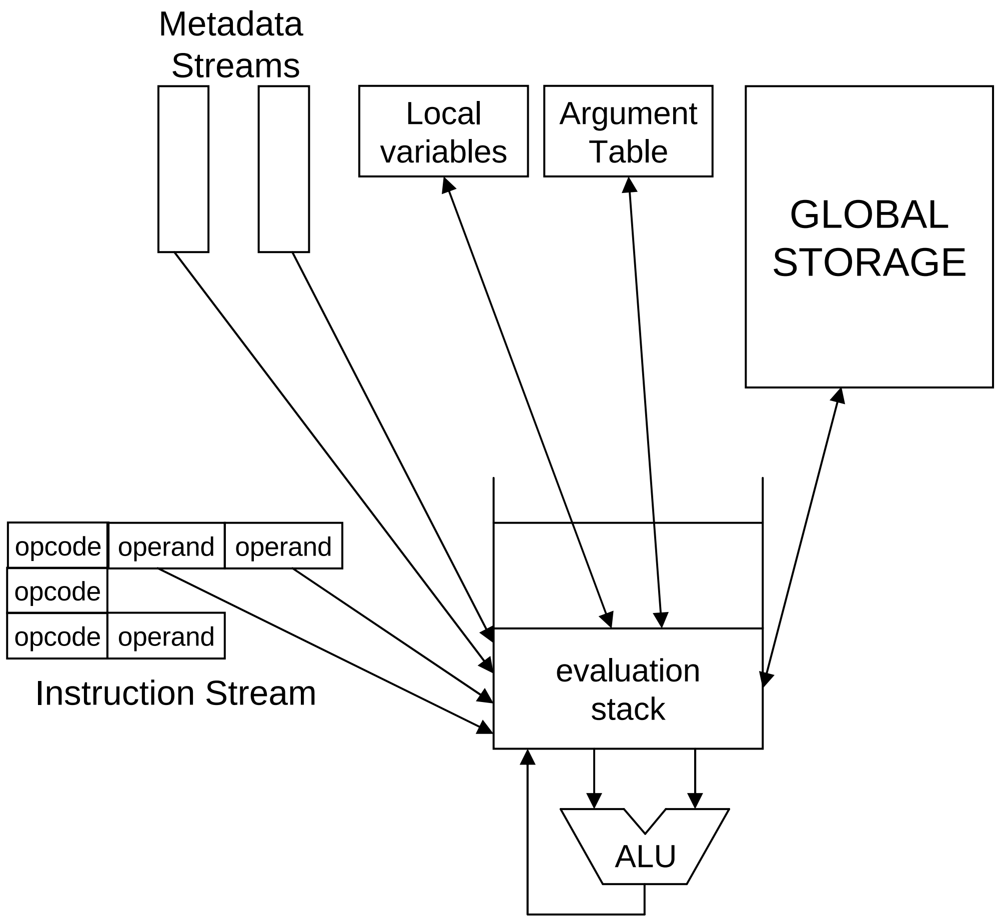
- Allow programmer to pass information to runtime via metadata
- Running program can access it by reflection

```
using System;
public class MainApp
{
    public static void Main()
    {
        // This generates a compile-time warning.
        int MyInt = Add(2,2);
    }

    // Specify attributes between square brackets
    // in C#. This attribute is applied only to
    // the Add method.
    [Obsolete("Will be removed in next version")]
    public static int Add( int a, int b)
    {
        return (a+b);
    }
}
```

# MSIL

- Stack-oriented V-ISA which is similar to bytecode
- Support constant pool called *stream*
- Control flow instruction
  - Verifiable PC relative jump or call/return.
  - Supports unverifiable function pointer
- Only for vector instruction (no array)
  - APIs for array
- Static type checking based on stack tracking
- Not a typed instruction set (`add` instead of `addi`)

# MSIL Memory Architecture



# Example

Java code

```
class Retangle {
    protected int sides [];

    ...

    public int perimeter() {
        return 2*(sides[0] + sides[1] );
    }

    ...

}
```

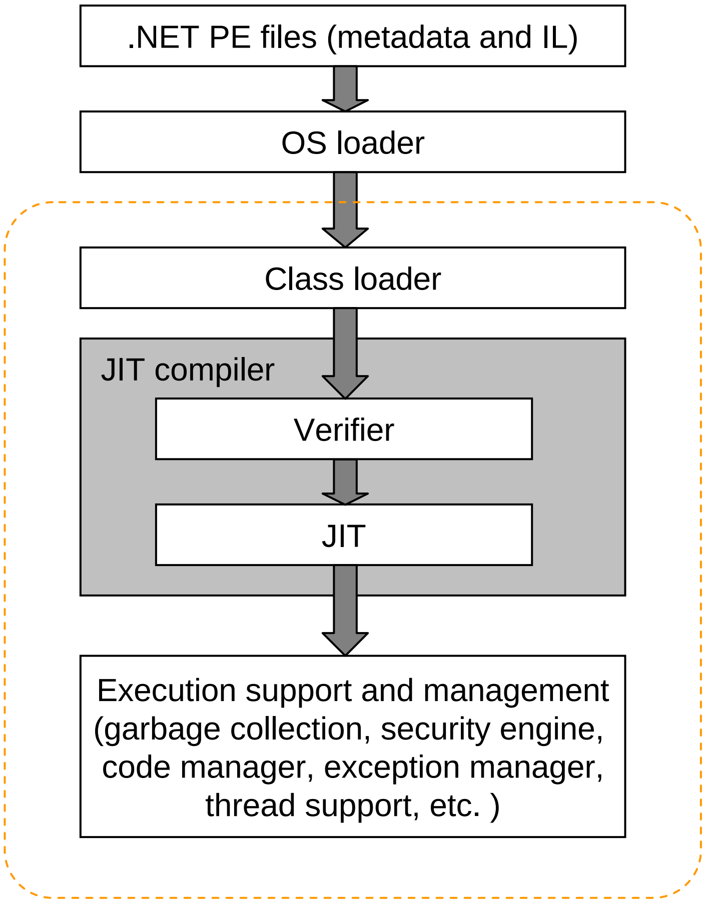
Java bytecode

```
0:  iconst_2
1:  aload_0
2:  getfield #2
5:  iconst_0
6:  iaload
7:  aload_0
8:  getfield #2
11: iconst_1
12: iaload
13: iadd
14: imul
15: ireturn
```

MSIL code

```
0:  ldc.i4.2
1:  ldarg.0
2:  ldobj <token>
5:  ldc.i4.0
6:  ldelem.i4
7:  ldarg.0
8:  ldobj <token>
11: ldc.i4.1
12: ldelem.i4
13: add
14: mul
15: ret
```

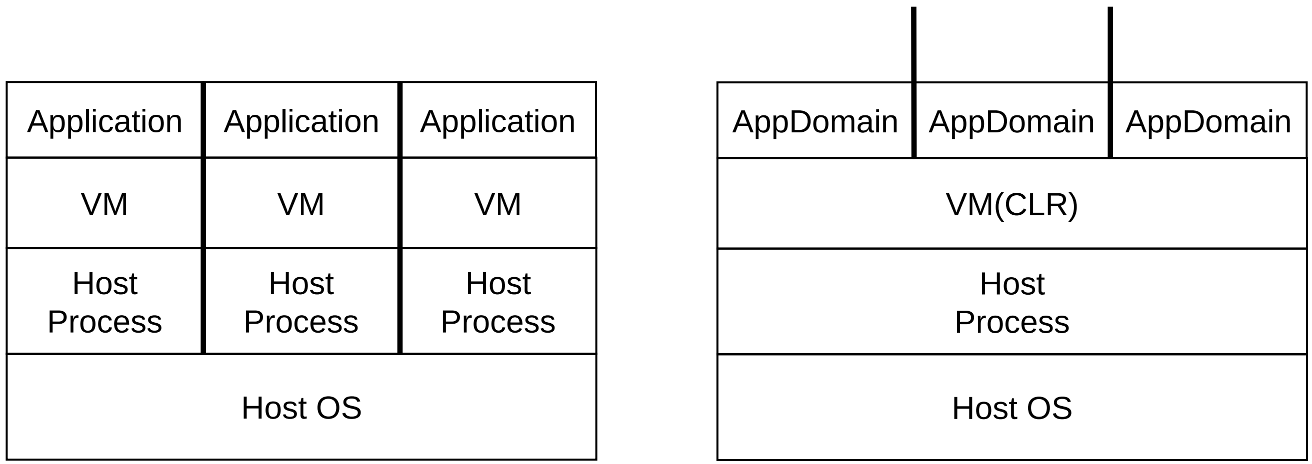
# CLR Execution



CLR's Virtual Execution Engine

- Verifier
- JITC compiler
  - JIT compilation occurs only the first time a method is invoked
  - Ngen pre-JITC during installation and setup time

# Isolation and AppDomains




- CLI supports running multiple application on a VM
  - Useful for increasing system utilization
  - But should be careful about security
  - Not very successful in Java
- AppDomains (application domains)
  - Share same VM with complete, lightweight isolation
  - Each process may contain multiple threads

# Summary: HLL VM vs. Process VM

- Metadata in V-ISA allows verification/interoperability
  - Regular ISA does not have it. Compiler use data structures and then throw away; they are implicit in binary code
- Memory architecture is more abstract
- Strict limitation in memory address formulation
- Relaxed requirement for precise exception
- No registers lead to simpler emulation
- Instruction discovery is obvious
- No self-modifying or self-referencing code
- Minimal OS dependence thru libraries