Compiler Construction: Practical Introduction

System Course for SRR Engineers

Samsung Research Russia
Moscow 2019

Lecture 8
Virtual Machines

- Virtual Machine
- JVM vs .NET VM
- Java Virtual Machihe
- .NET Virtual Machine

Virtual Machine: The Idea

- A Virtual Machine (VM) is software implementation of a machine (for example, a computer) that executes programs like a physical machine.
- Virtual Machines are separated into to major classification:
  - System Virtual Machines (also termed Full Virtualization VMs) provide a substitute for a real machine. They provide functionality needed to execute entire operating systems.
  - Process Virtual Machines are designed to execute computer programs in a platform-independent environment.

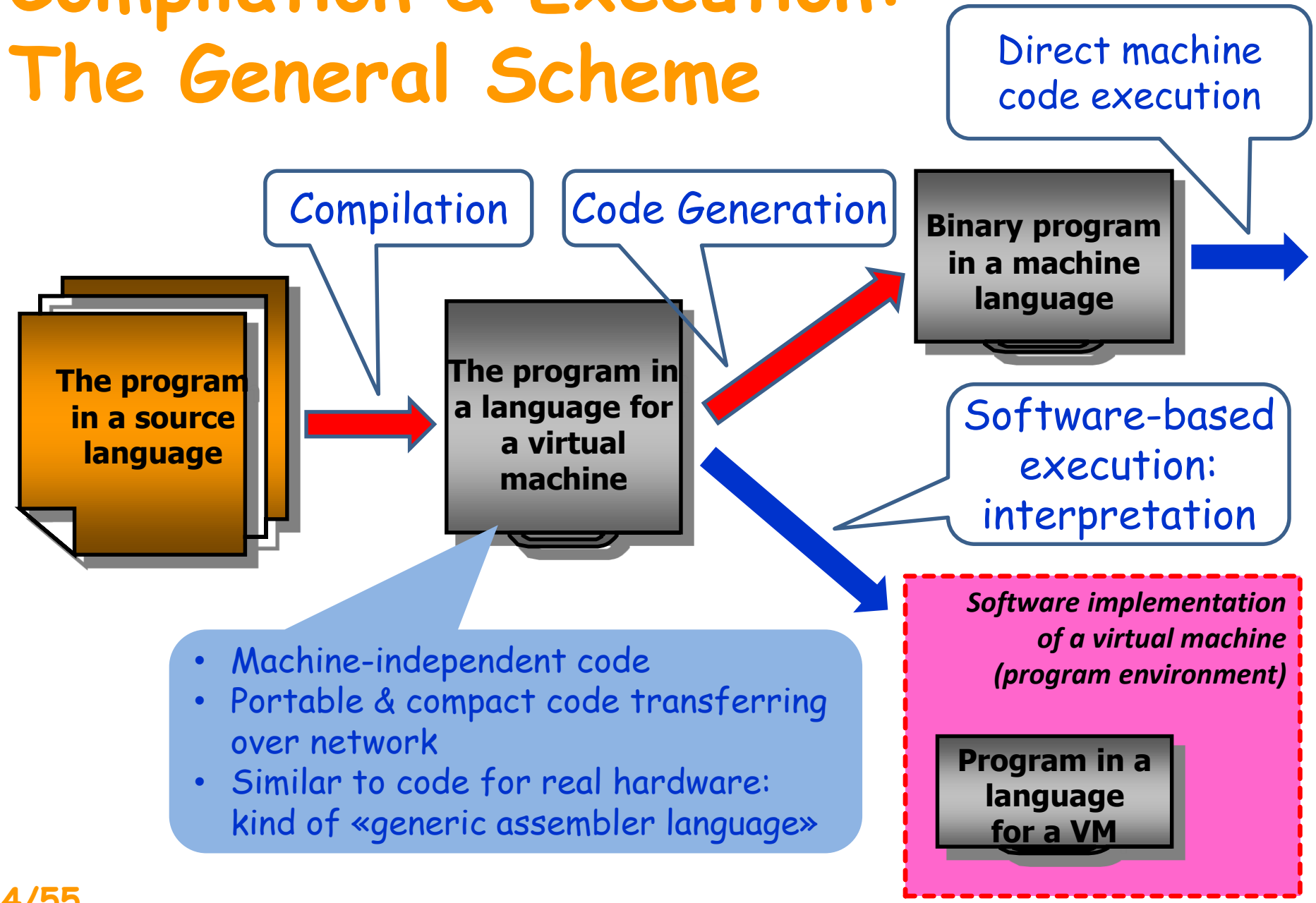
Virtual Machine: The Idea

The source program gets compiled…

- Neither to an object code (or an executable program) for a particular hardware architecture;
- Nor to an intermediate representation carrying information about source code semantics -

- But to a program for some hypothetical (abstract, virtual) computer with all architectural features of a real computer: a "CPU" with instruction set, with memory, registers etc.

Compilation & Execution: The General Scheme



Virtual Machine: What's New?

What's the real difference between conventional *program intermediate representation* and virtual machine code??

- Virtual machine language is designed not for adequate and complete **semantic representation** of the source program (as IR), but for portability and for program **execution**.
- Virtual machine architecture is made quite **similar to real hardware architecture**.

Brief History

- Snobol-4: The language for symbolic manipulations: 1967 (!!!)

  Snobol-4 programs translated into the code for SIL (System Implementation Language) abstract machine

- N.Wirth's Pascal compiler: 1973 (!!)

  Pascal source programs get compiled to code of an abstract Pascal machine: **P Code**.
  The next generation was **M Code** for Modula-2 language and its compiler.

- **Java Virtual Machine (JVM)**
  .NET Platform

- Python language

  Has its own abstract machine

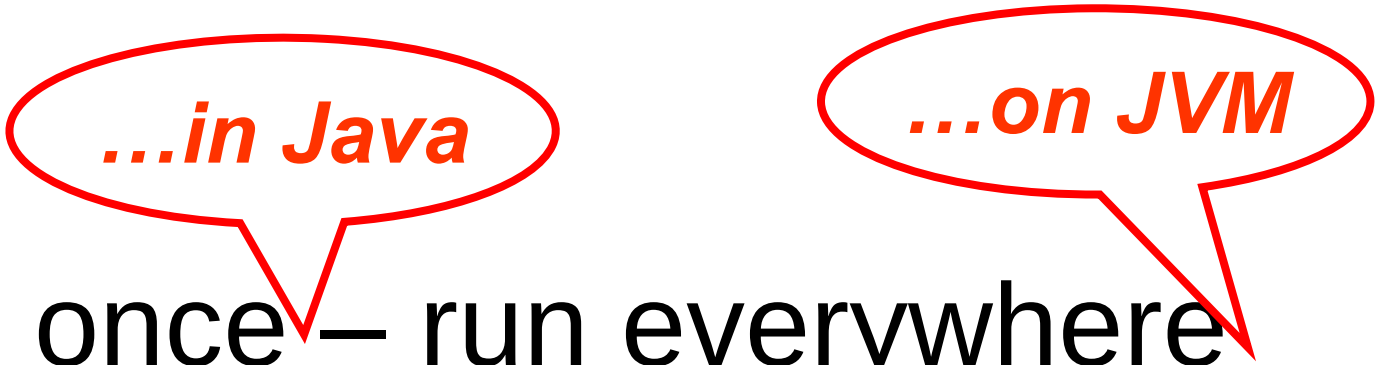
JVM & .NET: major features

from compiler writers' point of view ☺

- JVM & .NET are Process Virtual Machines
- Hardware independence
  - however, rather "close" to real machines
- **Stack-based execution model**
  - not only function calls, but expression calculations as well
- Rather high level of the instruction set
  - high-level function call mechanism; exception mechanism is supported
- Advanced code structure
  - constants, metadata (!), debug information
- Open format:
  - ISO standard for .NET, complete documentation for JVM

JVM & .NET: Philosophy

Java Slogan:



Write once – run everywhere

.NET Slogan:

Write in any language –
run under .NET

JVM & .NET: Comparison (1)

- Official Java/JVM slogan:
  Write once – run everywhere
  (but only under JVM ☺)

  The single language and many hardware platforms

- (Unofficial) .NET slogan:
  Write for .NET in any language – and get full interoperability (but only for Windows ☺)

  Many languages – the single platform (Windows)

JVM & .NET: Comparison (2)

- **Implementation** :
  JVM: **many** implementations (Sun/Oracle was just the first) for several hardware architectures.
  .NET: at least **four** implementations:
  the two of Microsoft («main version» и Rotor which is open source), **Mono** & Portable.NET.

- **Supported source languages**:
  Many (other that Java) for JVM.
  Many (other than C#) for .NET.

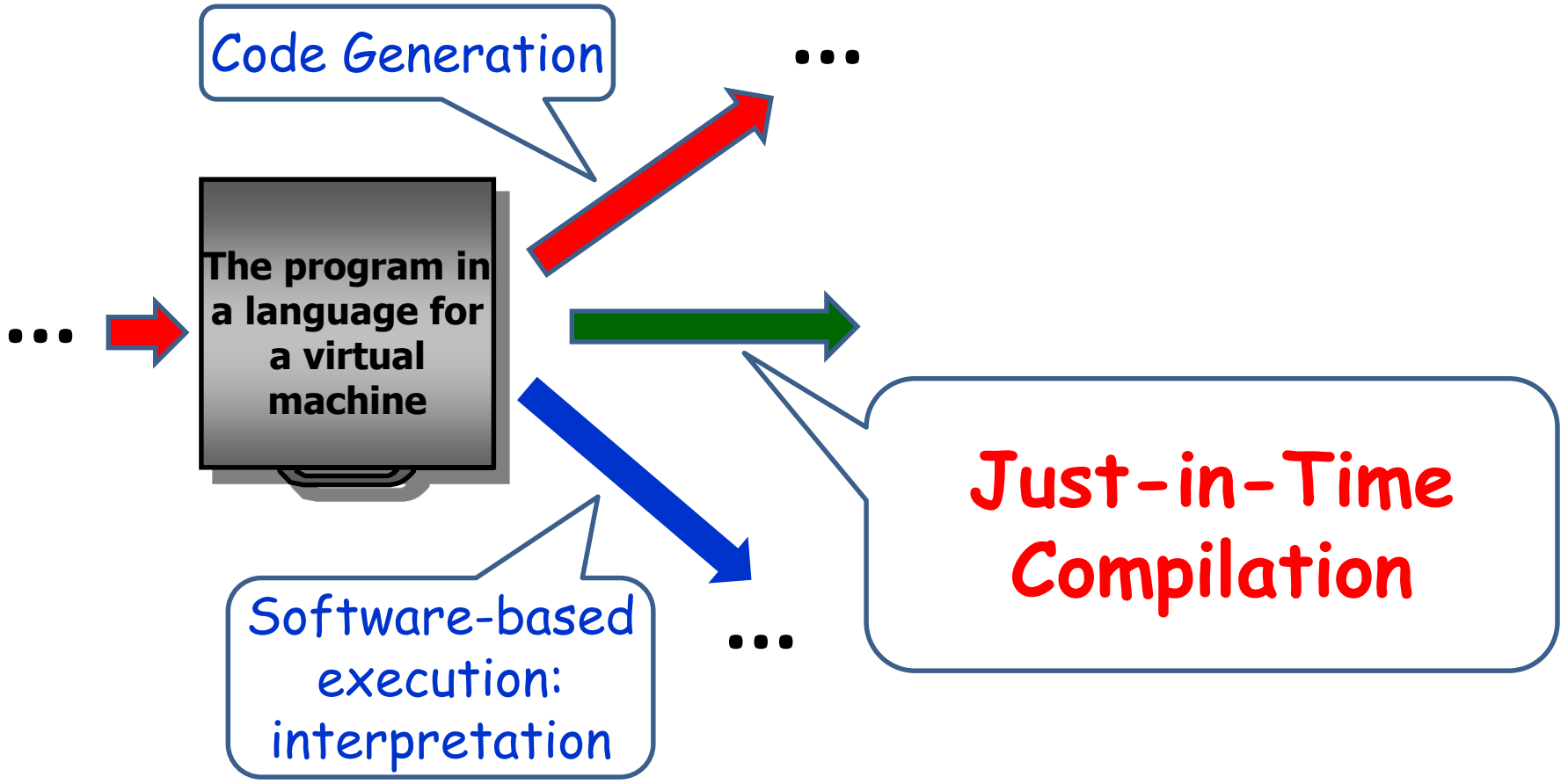
JVM & .NET: Comparison (3)

Standardization:

- Neither Java, nor Java Virtual Machine **are not yet standardized**.

- Not only *C#* language, but all .NET platform components (architecture, type system, instruction set, common language infrastructure etc.) – are **standardized** by both ECMA (European standard organization), and by ISO (International Standard Organization).

Compilation & Execution: Addition to the Common Scheme - JIT



Java Virtual Machine (JVM)

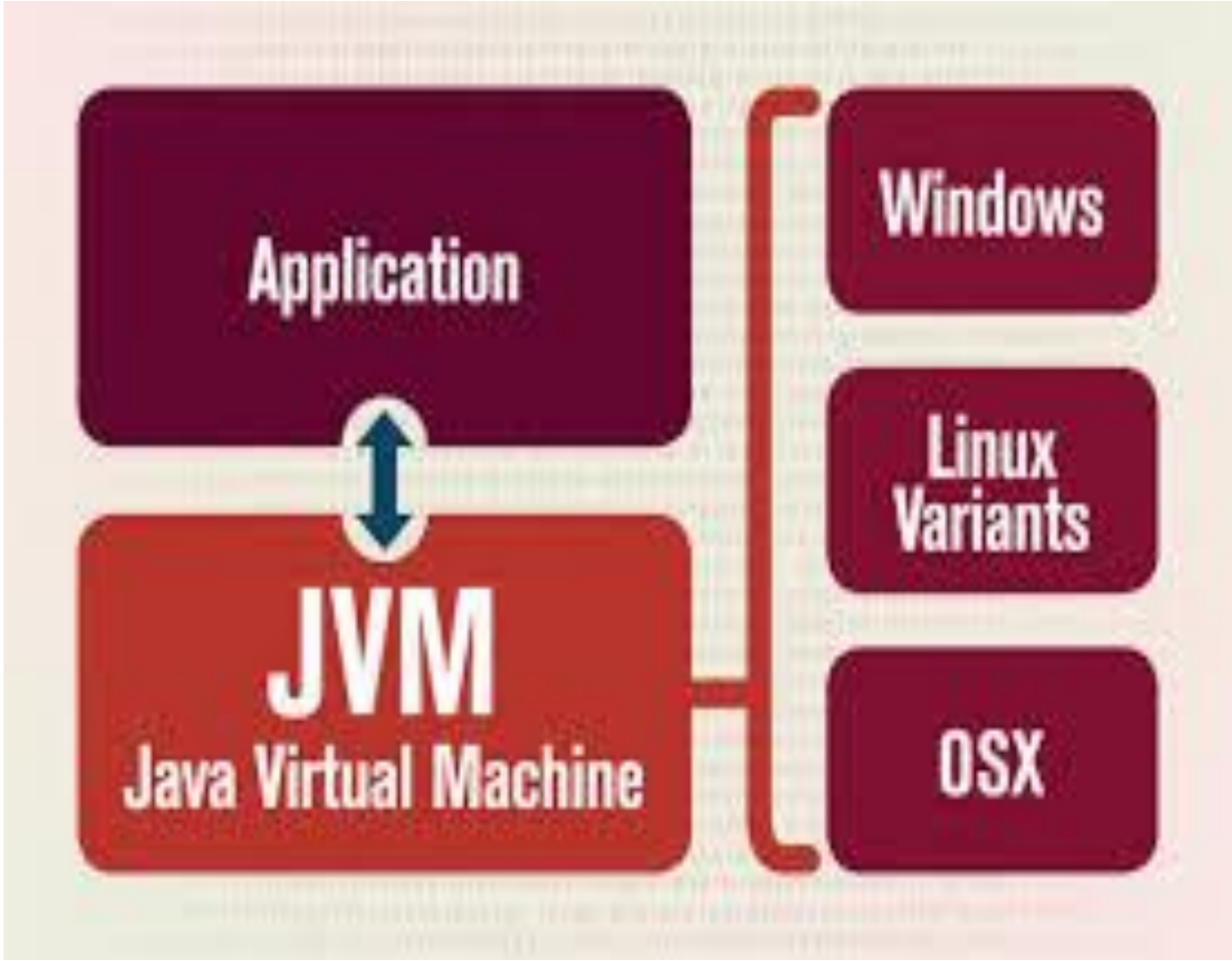
- A Java Virtual Machine (JVM) is a Process Virtual Machine that can execute Java Byte Code.
- JVM is converts Java Byte Code into Machine Language and executes it.
- JVM is platform independent: JVMs are available for many hardware and software platforms.
- JVM gives the flexibility of platform independence.
- JVM enables a set of computer software programs and data structures to use a virtual machine model for the execution of other computer programs and scripts:
  - Not just Java and Java-clone languages now supports many languages: Ada, C/C++, Lisp, Python.

Why Java Virtual Machine?

SAMSUNG RESEARCH

- A Java platform was initially developed to address the problems of building software for networked consumer devices.
- It was designed to support multiple host architectures and to allow secure delivery of software components.
- To meet these requirements, compiled code had to survive transport across networks, operate on any client, and assure the client that if was safe to run.
- "Write Once, Run Anywhere".

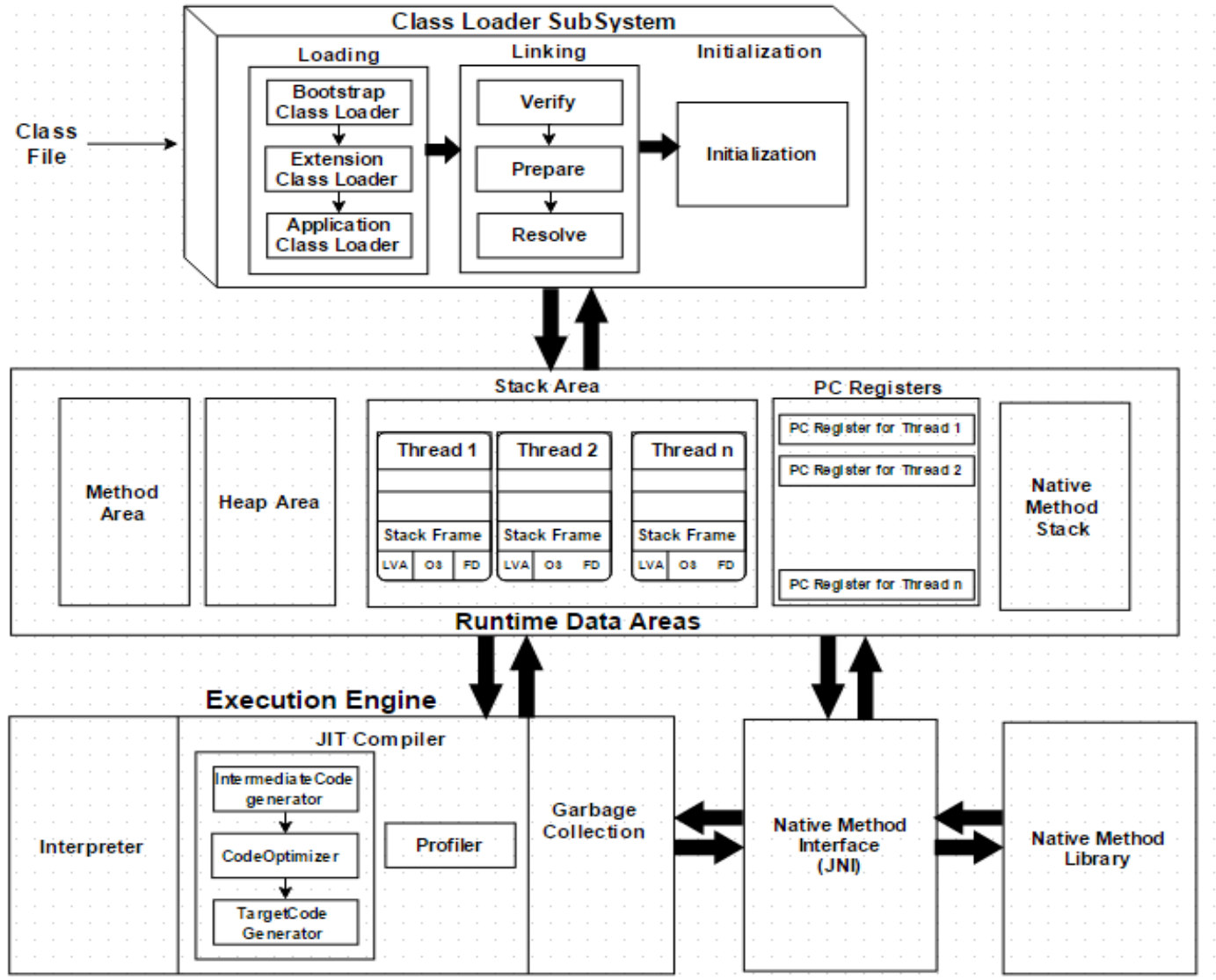
Write Once Run Anywhere



Java Principles

- Sun Microsystems set five primary goals in the creation of the Java language:
  - It should be "simple, object oriented, and familiar".
  - It should be "robust and secure".
  - It should be "architecture neutral and portable".
  - It should execute with "high performance".
  - It should be "interpreted, threaded, and dynamic".

Java Run-Time System



Java Class File

SAMSUNG RESEARCH

- Java compilers generate class file:
  - Magic number (`0xCAFEBABE`).
  - Minor version/major version.
  - Constant pool.
  - Access flags.
  - This class.
  - Super class.
  - Interfaces.
  - Fields.
  - Methods.
  - Attributes (extra hints for the JVM or other applications).

Java Class Loading

- Classes are loaded laizily when first accessed.
- Class name must match file name.
- Super classes are loaded first (transitively).
- The bytecode is verified.
- Static fields are allocated and given default values.
- Static initializers are executed.

Java Virtual Machine Principles

- Target Hardware – all CISC and RISC.
- Machine Type – Stack Machine.
- "Big Endian" encoding – large order bits in the lower address.
- Instructions are byte aligned for memory efficiency.
- Instructions are closely related to Java sources.

Java Virtual Machine Registers

SAMSUNG RESEARCH

- `pc` – Program Counter.
- `optop` – Pointer to the top of the operand stack.
- `frame` – Pointer to the current execution environment.
- `vars` – Pointer to the first (0^{th}) local variable in the current execution environment.

JVM Instruction Set Architecture

SAMSUNG RESEARCH

- JVM instruction consists of a one-byte opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation.
- Operands are not required, there are many instructions that consist of only the opcode.
- One-byte instructions allow for up to 256 instructions.
- Each instruction has a mnemonic name which is mapped to the binary one-byte opcode.

Java Bytecode Instructions

SAMSUNG RESEARCH

- The JVM ISA (Instruction Set Architecture) is a CISC architecture.
- The JVM has 256 instructions for (see the JVM specification for the full list):
  - Arithmetic and logic operations.
  - Type conversion.
  - Branch operations (control transfer).
  - Constant loading operations.
  - Local operations (load and store).
  - Stack operations (stack operand management).
  - Class and object operations (creation and manipulation)
  - Method operations (invocation and return).

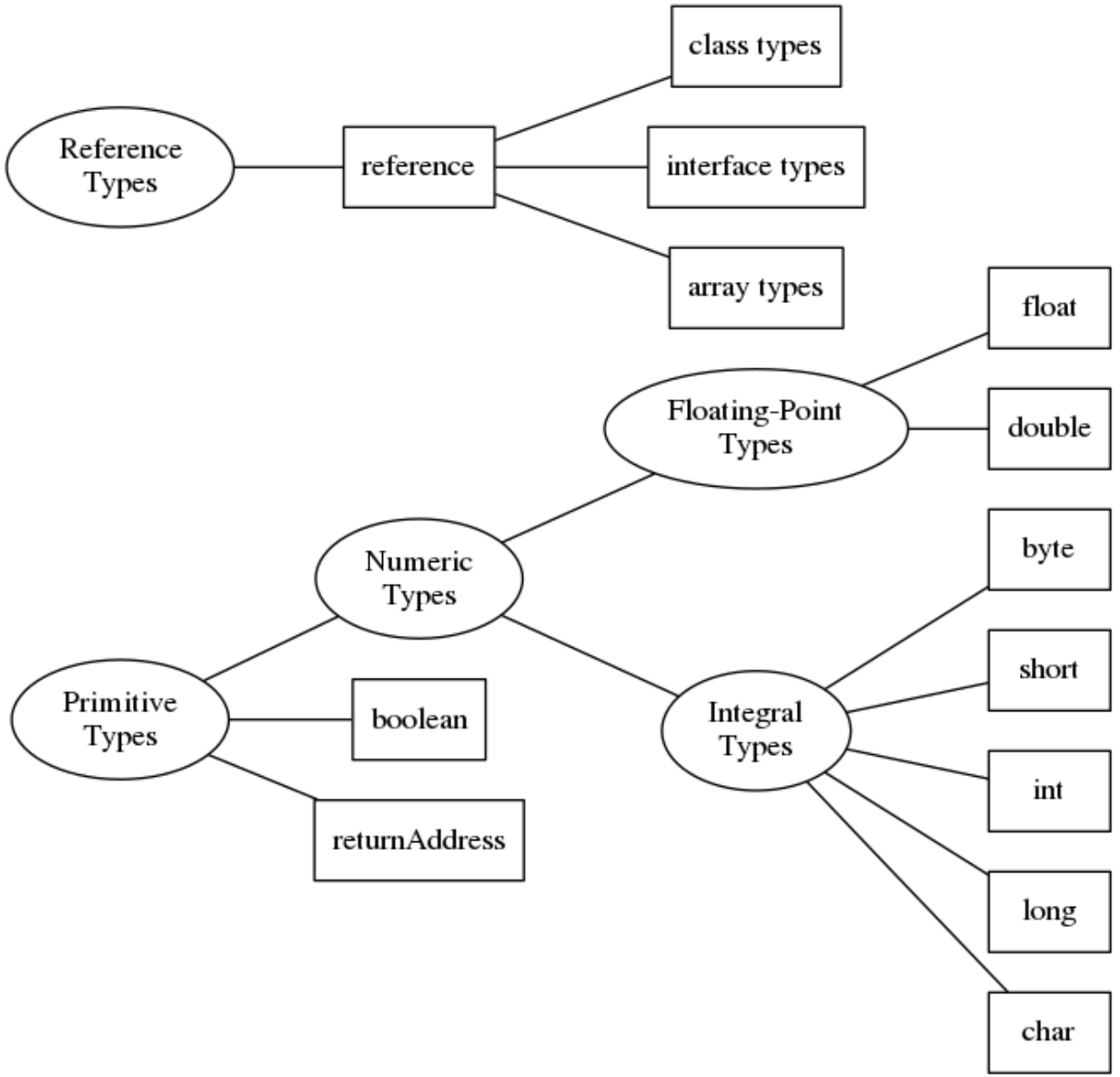
JVM Data Types

- The JVM operates on two kinds of types: primitive types and reference types.
- Integral Types:
  - Byte – 8bit signed integers.
  - Short – 16bit signed integers.
  - Int – 32bit signed integers.
  - Long – 64bit signed integers.
  - Char – 16bit unsigned integers representing Unicode charecters.
- Floation Point Types:
  - Float – 32bit single-precision float.
  - Double – 64bit single-precision float.
- Boolean – values true and false.
- returnAddress – pointers to the opcodes of JVM instructions.

JVM Data Types

- Three kinds of reference types:
  - Class types.
  - Array types.
  - Interface types.
- These reference dynamically created classes, arrays or interface implementations.
- Can be set to `null` when not referencing anything and then cast to any type.

JVM Data Types

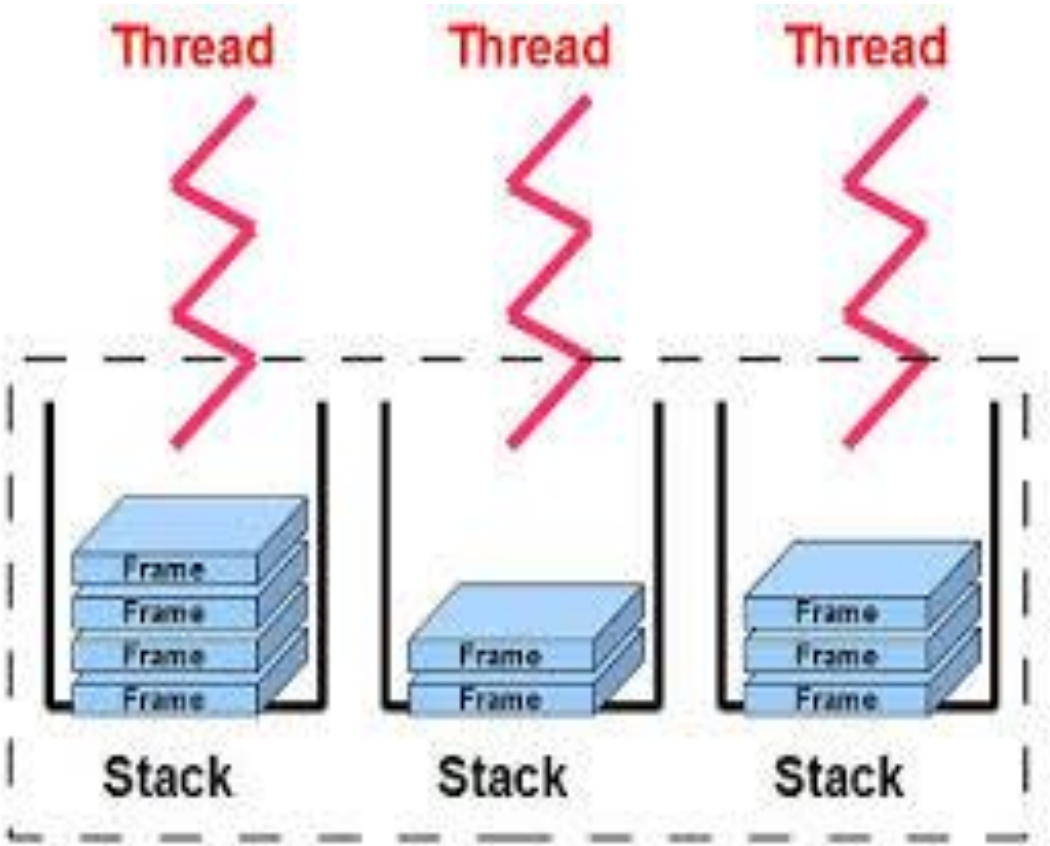


JVM Stack Frame

SAMSUNG RESEARCH

- As threads are created each thread get a Java Stack and pc register.
- JVM creates a stack frame for each method of a class. Each method Stack Frame consists of:
  - Local Variables.
  - Execution Environment.
  - Operand Stack.

JVM Stack Frame



JVM Stack Frame

SAMSUNG RESEARCH

- **Local Variables** form array of 32-bit variables:
  - Types longer than 32-bit (double) use consecutive cells.
  - Pointed at by **vars** register.
  - Loaded onto and stored from the operand stack.
- **Execution Environment** is info about the current state of the JVM stack and consists of:
  - Pointer to the previous invoked method frame.
  - Pointer to the local variables (**vars** register).
  - Pointer to the of the operand stack.
  - Pointer to constant pool
  - Pointer to the method code
- **JVM Operand Stack**:
  - Is 32-bit **LIFO**.
  - Holds the argument for opcodes.
  - Is a subsection of the JVM Stack: primary area for the current status of bytecode execution.

JVM Operand Stack

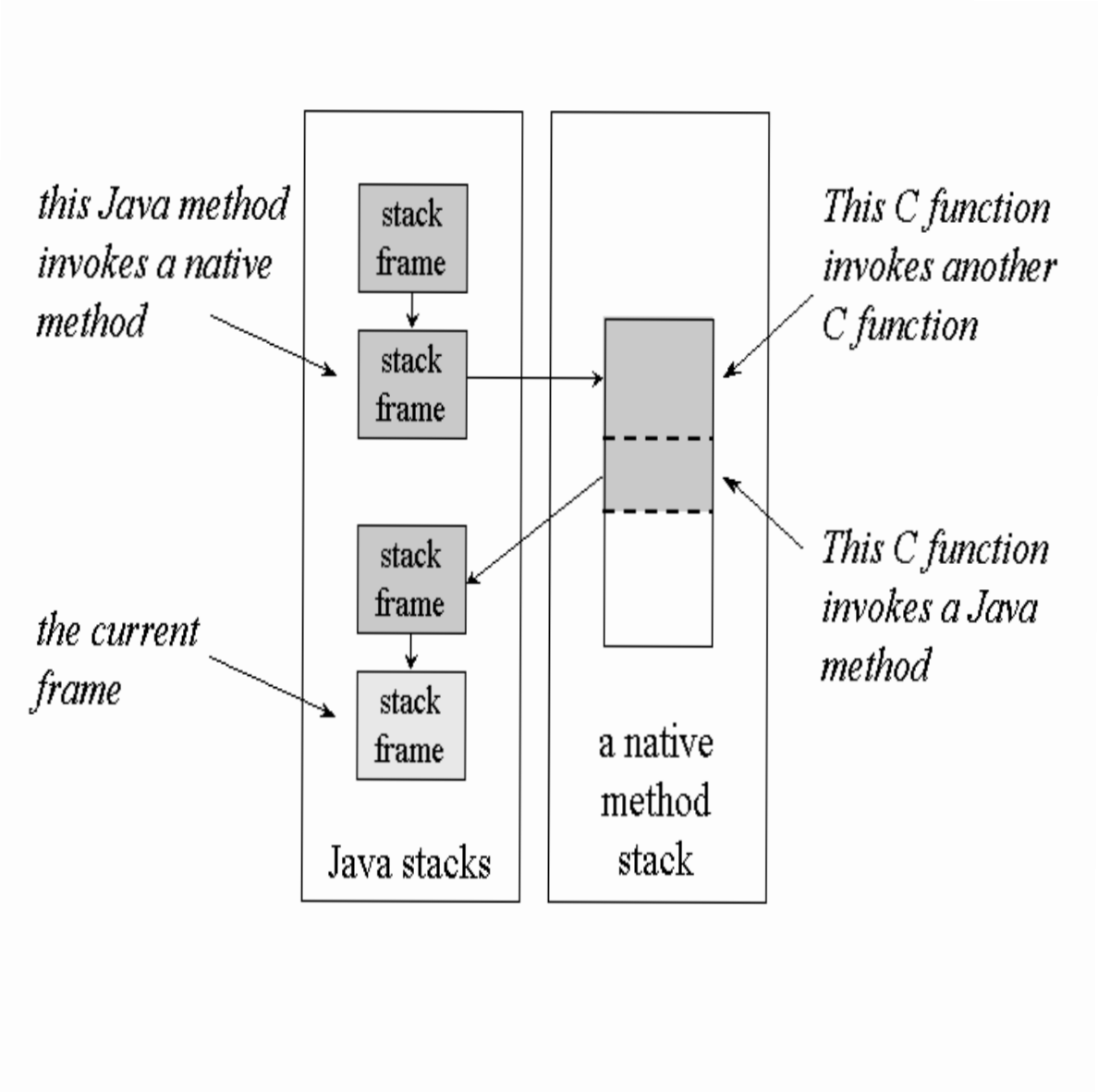
Offset	Bytecode Instruction	Stack before	Stack after	Size
0	iload_1	<empty>	I a	1
1	iload_2	I a	I a, I b	2
2	if_icmple	I a, I b	<empty>	0
	true			
5	iload_1	<empty>	I a	1
6	iload_2	I a	I a, I b	2
7	iadd	I a, I b	I (a+b)	1
8	goto	I (a+b)	I (a+b)	1
	false			
11	iload_1	<empty>	I a	1
12	iload_2	I a	I a, I b	2
13	isub	I a, I b	I (a-b)	1
14	ireturn	I (a+b) | I (a-b)	<empty>	0

JVM Execution Engine

SAMSUNG RESEARCH

- The bytecode, which is assigned to the Runtime Data Area, will be executed by the Execution Engine, which is consisted from:
- Interpreter - interprets the bytecode faster but executes slowly. When every method called multiple times, is interpeted anew.
- JIT Compiler – when Execution Engine finds repeated code it uses JIT compiler, which compiles the entire bytecode and translate it to native code.
  - Intermediate Code Generator – produces intermediate code.
  - Code Optimizer – optimize the intermediate code generated above.
  - Target Code Generator – produces machine (native) code.
  - Profiler – a special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
- Garbage Collector – collects created objects and removes unreferenced ones.
- Java Native Interface (JNI) & Native Method Libraries – is a foreign function interface programming framework that enables Java code running in a Java virtual machine (JVM) to call and be called by native applications (specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly..

Native Method Stack

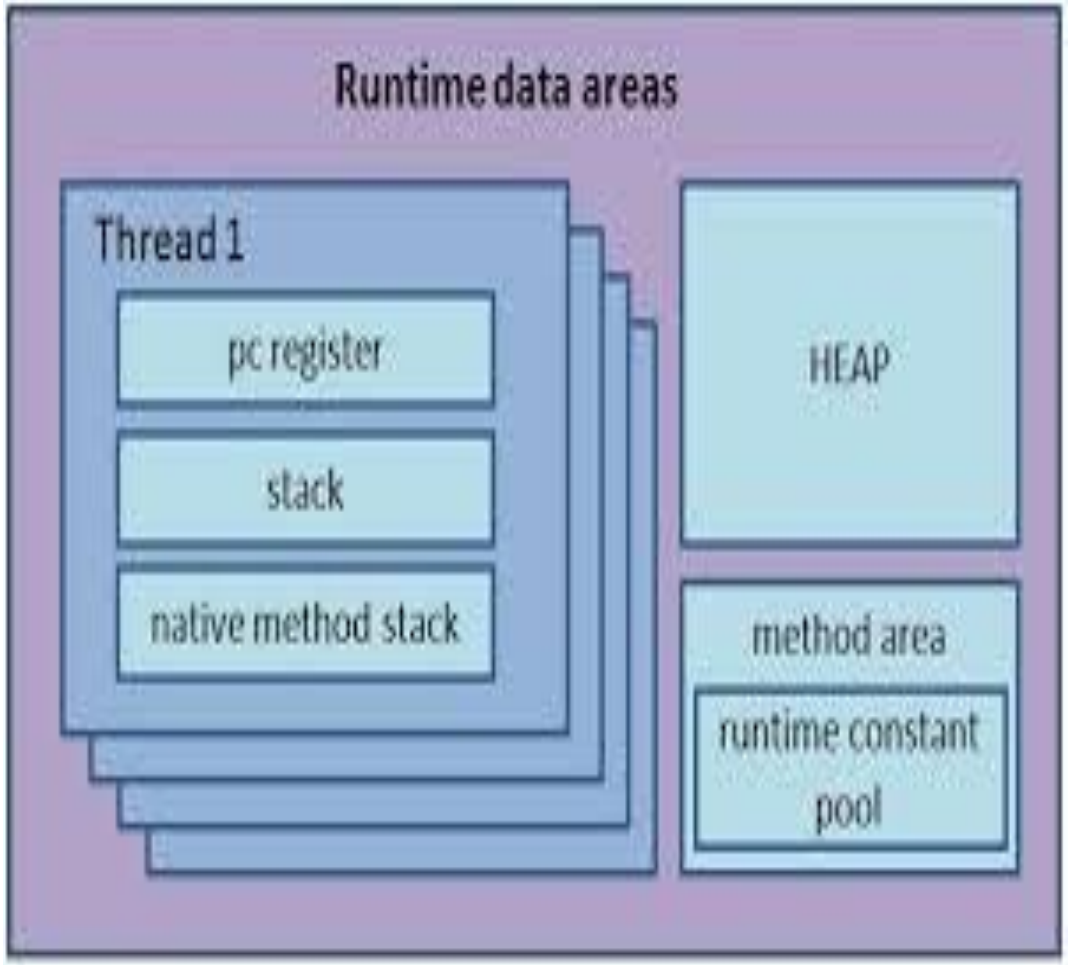


JVM Memory

SAMSUNG RESEARCH

- **JVM Garbage Collected Heap**:
  - Memory from which class instances are allocated.
  - Interpreter monitors memory usage and reclaims memory when no longer in use.
  - Garbage Collection is automatic.
- **Method Area** - bytecodes for all Java Methods.
- **Constant Pool** – class names, method and fields names, strings, constants.
- **JVM Limitaions**:
  - 4GB internal addressing but to 32bit wide stack implementation.
  - Methods are limited to 32Kb due to 16bit offset addressing used for branching.
  - 256 local variables/stack (8bit field).
  - 32KB constant pool entries for a method.

JVM Memory


Runtime data areas
Thread 1
pc register
stack
native method stack
HEAP
method area
runtime constant pool

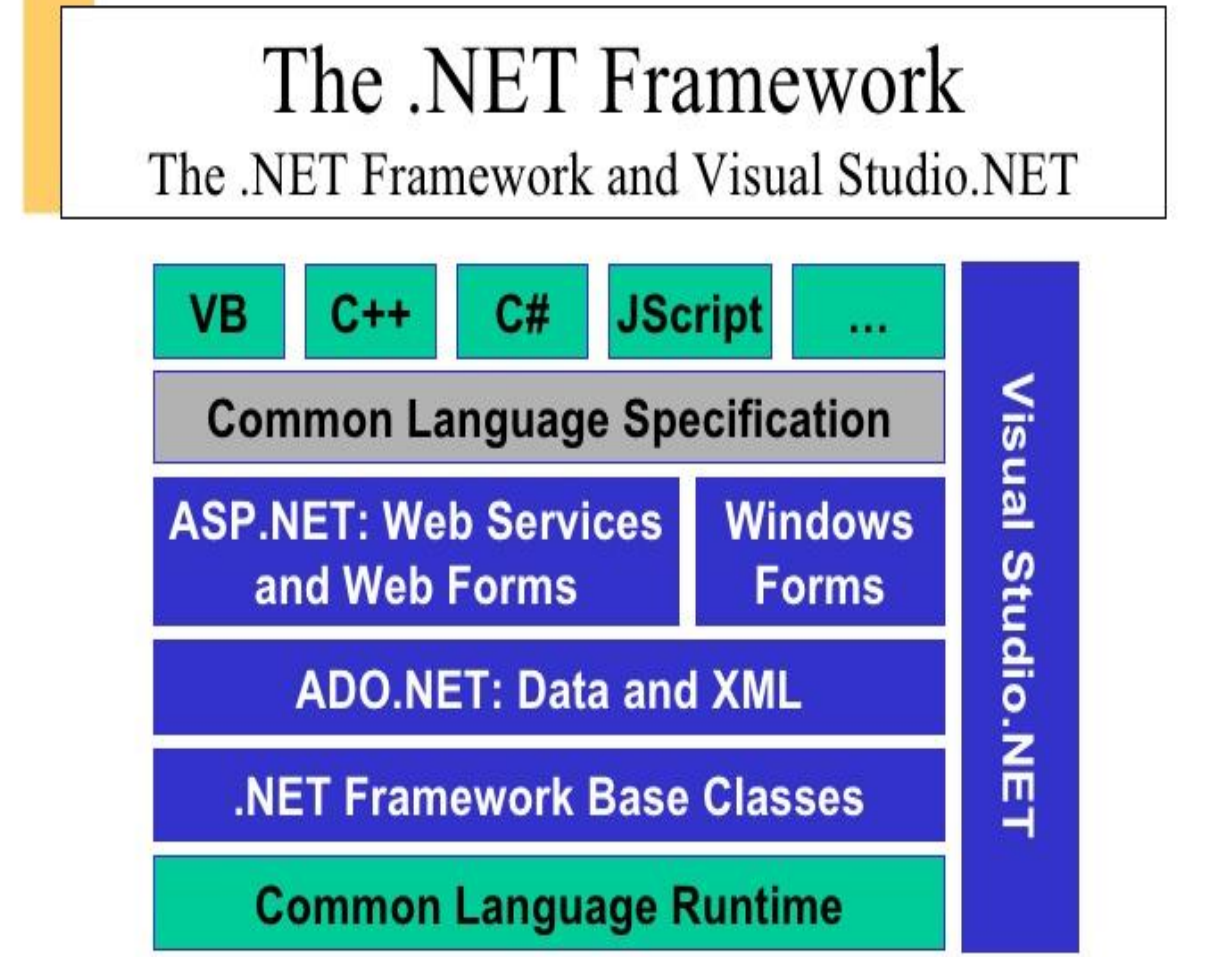
What Is .NET Framework

- The infrastructure for the overall .NET Platform.
- A computing platform designed to simplify application development.
- A consistent object-oriented programming environment.
- A code-execution environment that minimizes software deployment and versioning conflicts.

Major Components of .NET

- Common Language Runtime (CLR)
  - Manages code execution at runtime
  - Manages memory, threads, etc
  - Code designed for the CLR is referred to as **Managed Code**
    - Object oriented
    - Cross-language integration
    - Cross-language **exception handling**
    - Multiple version support
- Base Class Library (Framework Class Library – FCL)
  - Object-oriented collection of reusable types
  - Sits on-top of the Common Language Runtime (CLR)
- Common Type System (CTS)
- Common Language Specification (CLS)

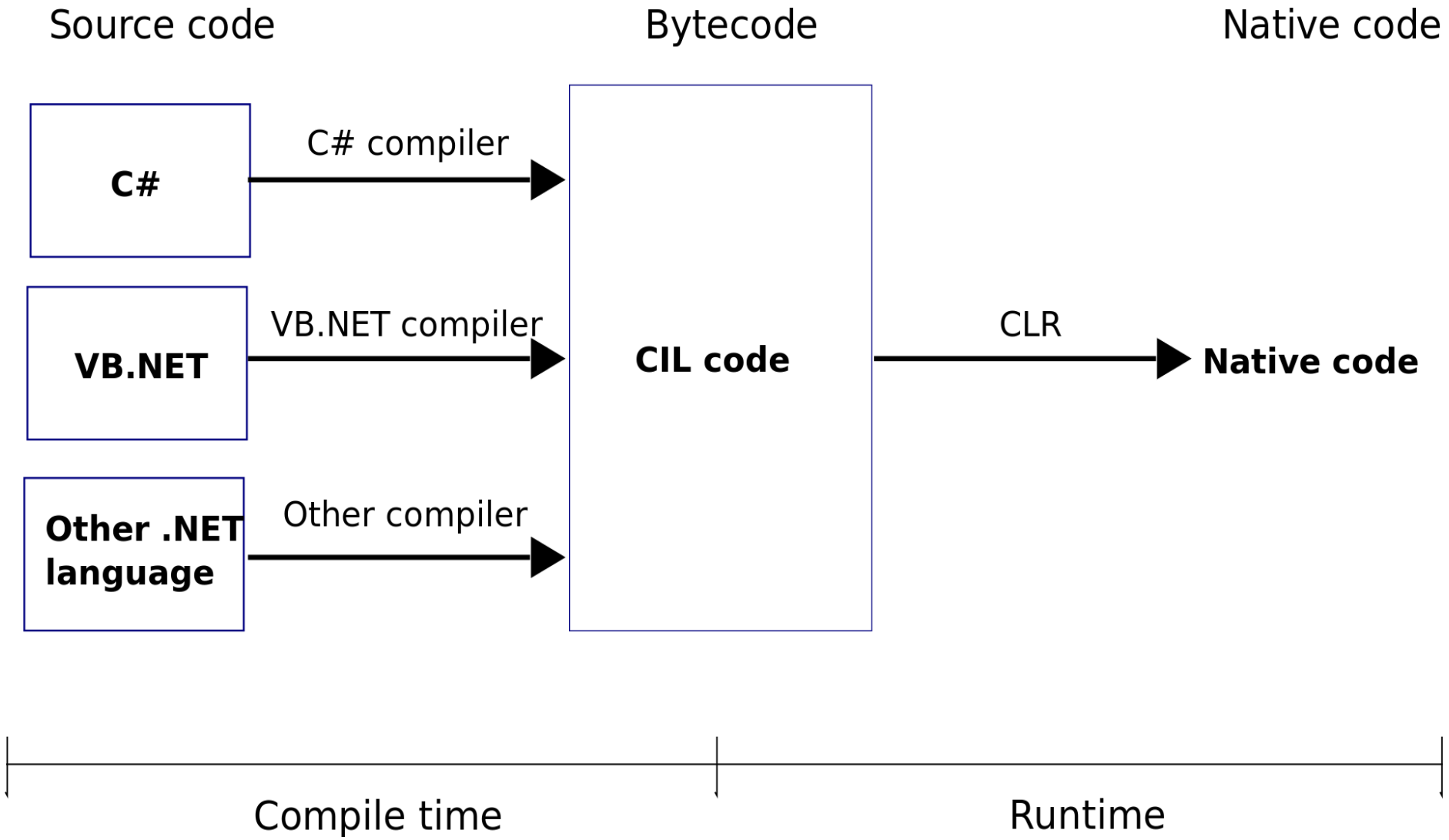
.NET Framework Structure



Common Language Runtime (CLR)

- Common Language Runtime (CLR) works like a virtual machine in executing all languages.
- Checking and enforcing security restrictions on the running code.
- Manages memory through an extremely efficient garbage collector.
- Common Type System (CTS).
- Conversion from IL into code native to the platform being executed on.

Common Language Runtime (CLR)



Common Language Runtime (CLR)

- CLR manages object layout and references to objects.
- Objects whose lifetimes are managed by the CLR are referred to as Managed Data.
- Automatic memory management reduces memory leaks.
- In Managed Code you can use:
  - Managed Data.
  - Unmanaged Data.
  - Both.

Common Language Runtime (CLR)

- All CLR compliant compilers use a common type system:
  - Allows for cross-language inheritance.
  - Passing object instances across language barriers.
  - Invoking methods across language barriers.
- Managed components exposes metadata.
- Metadata includes:
  - Resource components was compiled against.
  - Information about types and dependencies.
  - Signatures of each type's method.
  - Members that your code references.
  - Other runtime data for CLR.

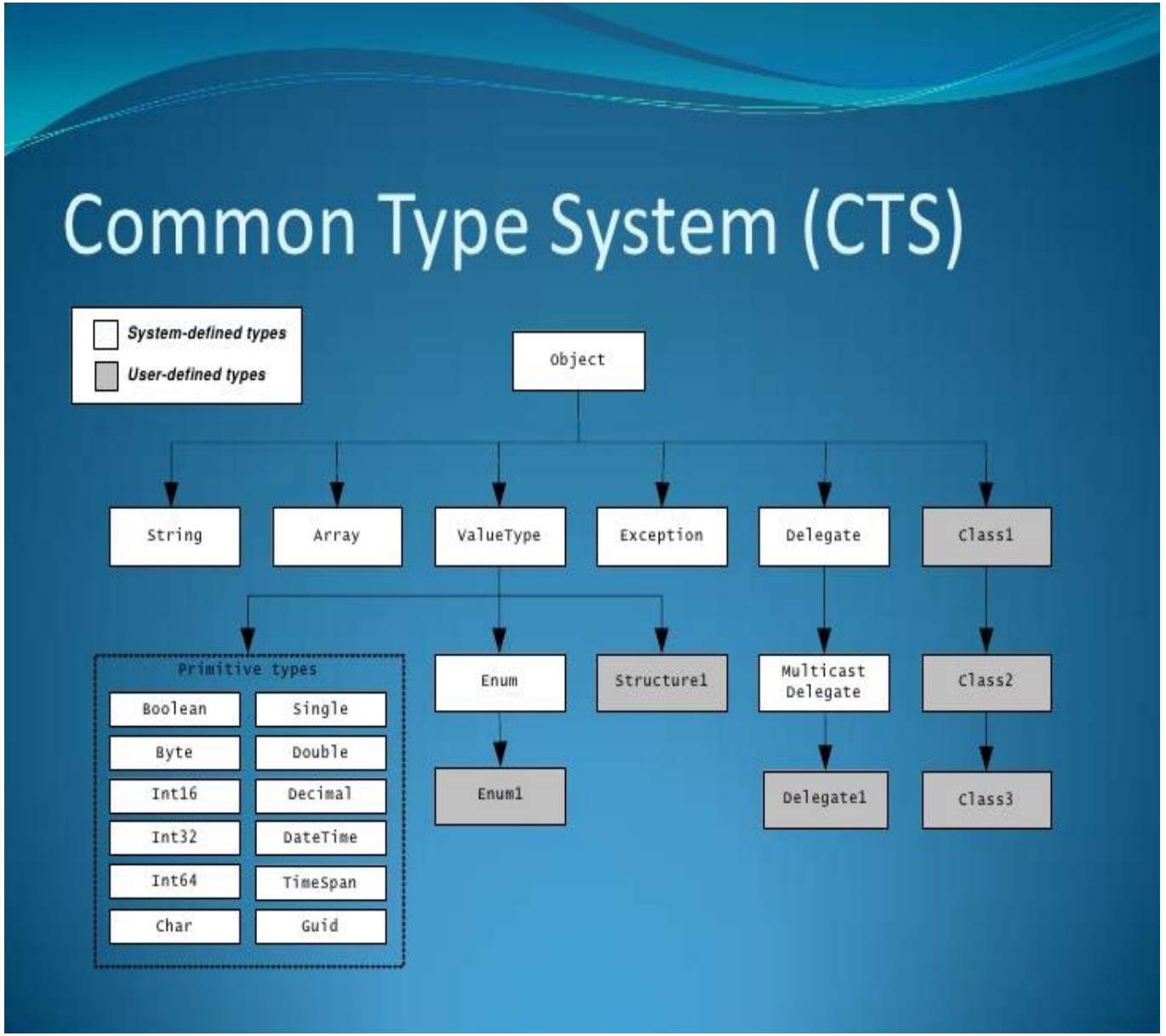
One Runtime for Many Languages

- CLR is on open standard.
- Any language can use of CLR services.
- Any language can use classes written in any other language.
- Any language can inherit classes written in any other language.

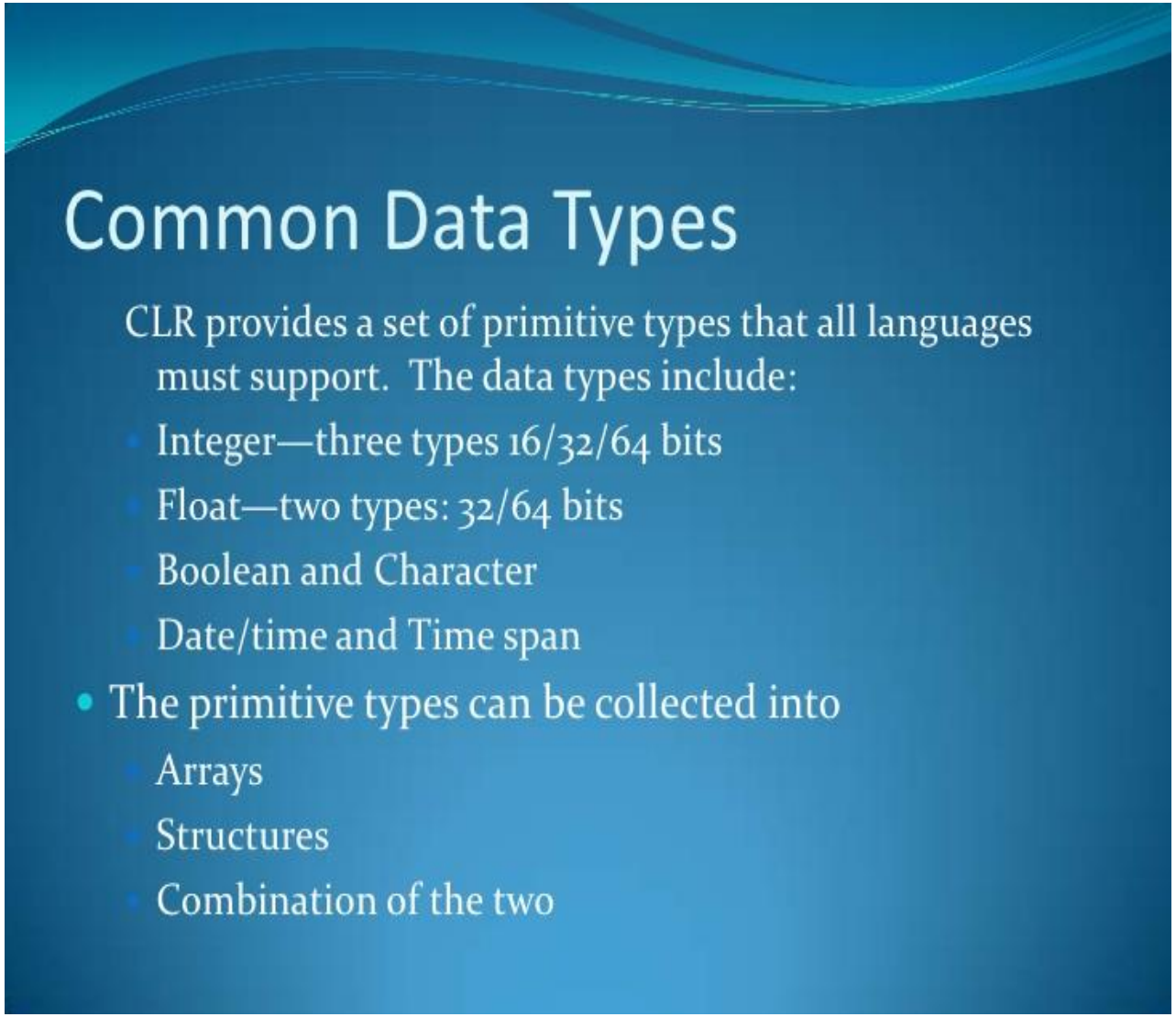
Common Type System (CTS)

- CTS is a rich type system built into the CLR:
  - Implements various types (`int`, `double`, etc.).
- Strictly enforces type safety.
- Ensures that classes are compatible with each other by describing them in a common way.
- Enables types in one language to interoperate with types in another language.

Common Type System (CTS)



Common Data Type



Common Language Specification(CLS)

- CLS is a set of specifications that language an library designers need to follow:
  - This will ensure interoperability between languages.
- Specification that a language must conform to, to be accepted into the .NET framework.
- The specification are detailed at https://www.ecma-international.org/publications/standards/Ecma-335.html

Intermediate Language (IL)

- .NET languages are not compiled to machine code. They are compiled to an Intermediate Language (IL).
- CLR accepts the IL code and recompiles it to machine code. The recompilation is Just-In-Time (JIT) meaning it is done as soon as a function or subroutine is called.
- The JIT code stays in memory for subsequent calls. In cases where there is not enough memory it is discarded thus making JIT process interpretive.

CIL Code Example

```
class Program
{
  int F(int a,int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

Even such a simple (actually trivial) code **is not optimal**; it can be made better. See the next lecture (optimization).

```
.method private hidebysig instance int32
        F(int32 a,
          int32 b) cil managed
{
  // Code size       17 (0x11)
  .maxstack  3
  .locals init ([0] int32 c,
           [1] int32 x,
           [2] int32 CS$1$0000)
  IL_0000:  nop
  IL_0001:  ldc.i4.7
  IL_0002:  stloc.0
  IL_0003:  ldarg.1
  IL_0004:  ldarg.2
  IL_0005:  sub
  IL_0006:  ldarg.1
  IL_0007:  ldloc.0
  IL_0008:  add
  IL_0009:  mul
  IL_000a:  stloc.1
  IL_000b:  ldloc.1
  IL_000c:  stloc.2
  IL_000d:  br.s       IL_000f
  IL_000f:  ldloc.2
  IL_0010:  ret
} // end of method Program::F
```

Microsoft Intermediate Language

- **Managed Code** is compiled into **Common Intermediate Language (CIL)**.
- CPU-independent set of instructions:
  - Loading, storing, initializing and caling methods.
  - Arithmetic and logical operations, etc.
  - Control flow, exception handling, direct memory access.
- Is **Object-Oriented**.
- Is **Stack-Based**.

Object-Oriented Concepts

- CIL may create objects, call methods and use other types of class members such as fields.
- CIL is designed to be Object-Oriented and every method (with some exception) needs to reside in a class.
- Instance classes:
  - An instance class contains at least one constructor and some instance members.
- CIL has instructions for creating objects.
- CIL has instructions for invoking instance methods.

CIL Instruction Groups

- CIL bytecode has instructions for the following groups of tasks:
  - Load and store.
  - Arithmetic.
  - Type conversion.
  - Object creation and manipulation.
  - Operand stack management (push, pop).
  - Control transfer (branching).
  - Method invocation and return.
  - Throwing exceptions.
  - Monitor-based concurrency.
  - Data and function pointers manipulation needed for C++ and unsafe C# code.

.NET Assemblies

- Assemblies are the smallest unit of code distribution, deployment and versioning.
- Individual components are packaged into unit called assemblies.
- Can be dynamically loaded into the executon engine on demand.
- Contains Common Intermediate Language (CIL) code to be executed.
- Security boundary – permissions are granted at the assembly level.
- Type boundary – all types include the assembly name they are a part of.

Minimal Hello Program in CIL

```
.assembly Hello {}
.method public static void Main() cil managed
{
  .entrypoint
  .maxstack 1
  ldstr "Hello, world!"
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Just-In-Time (JIT) Compiling

- **Assemblies** are compiled to native code by a **JIT compiler** before executing.
- Compiled assemblies include **metadata**.
- JIT compilers are built into the CLR for every supported CPU architecture.
- JIT compilers conver CIL to native on demand.
- Resulting native code is stored for reuse.
- JIT compiling occurs for each method after the application is restarted.

Classic Compiler vs .NET Compiler