Lecturer: Andrea Corradini
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-04: Runtime Systems and intro to JVM
Overview

- Runtime Systems
- The Java Runtime Environment
- The JVM as an abstract machine
- JVM Data Types
- JVM Runtime Data Areas
- Multithreading
- Per-thread Data Areas
- Dynamic Linking
- JIT compilation
- Method Area
Runtime system

• Every programming language defines an execution model
• A runtime system implements (part of) such execution model, providing support during the execution of corresponding programs
• Runtime support is needed both by interpreted and by compiled programs, even if typically less by the latter
Runtime system (2)

• The runtime system can be made of
  – Code in the executing program generated by the compiler
  – Code running in other threads/processes during program execution
  – Language libraries
  – Operating systems functionalities
  – The interpreter / virtual machine itself
Runtime Support needed for...

• Memory management
  – Stack management: Push/pop of activation records
  – Heap management: allocation, garbage collection

  ➔ Chapter 7 of "Dragon Book"

• Input/Output
  – Interface to file system / network sockets / I/O devices

• Interaction with the runtime environment,
  – state values accessible during execution (eg. environment variables)
  – active entities like disk drivers and people via keyboards.
Runtime Support needed for... (2)

- Parallel execution via threads/tasks/processes
- Dynamic type checking and dynamic binding
- Dynamic loading and linking of modules
- Debugging
- Code generation (for JIT compilation) and Optimization
- Verification and monitoring
Java Runtime Environment - JRE

• Includes all what is needed to run compiled Java programs
  – JVM – Java Virtual Machine
  – JCL – Java Class Library (Java API)
• We shall focus on the JVM as a real runtime system covering most of the functionalities just listed
• Reference documentation:
    https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
    https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf
New "short-term" releases of Java

- **Java 9**, released in September 2017
  - Added module system

- **Java 10**, released in March 2018
  - Type inference of local variables (using `var`)
New "short-term" releases of Java (2)

➔ **Java 11 (LTS)**, September 2018
  - Local parameter type inference for lambdas, using `var`
  - **ZGC**: A Scalable Low-Latency Garbage Collector (Experimental)
  - [https://openjdk.java.net/projects/jdk/11/](https://openjdk.java.net/projects/jdk/11/)

➔ **Java 12**, March 2019
  - `switch` expressions, preview
  - [https://openjdk.java.net/projects/jdk/12/](https://openjdk.java.net/projects/jdk/12/)

➔ **Java 13**, September 2019
  - [https://openjdk.java.net/projects/jdk/13/](https://openjdk.java.net/projects/jdk/13/)
What is the JVM?

- The **JVM** is an **abstract** machine in the true sense of the word.
- The JVM specification does *not* give implementation details like memory layout of run-time data area, garbage-collection algorithm, internal optimization (can be dependent on target OS/platform, performance requirements, etc.)
- The JVM specification defines a machine independent “**class file format**” that all JVM implementations must support
- The JVM imposes **strong syntactic** and **structural constraints** on the code in a class file. Any language with functionality that can be expressed in terms of a valid class file can be hosted by the JVM
Execution model

• JVM is a *multi-threaded stack based machine*

• JVM instructions
  • implicitly take arguments from the top of the operand stack of the current frame
  • put their result on the top of the operand stack

• The operand stack is used to
  • pass arguments to methods
  • return a result from a method
  • store intermediate results while evaluating expressions
  • store local variables
Java Abstract Machine Hierarchy

Java Application

Java Programming Language

Java Class Library

Java Virtual Machine
  - Classloader
  - Verifier
  - Execution

Operating System
Class Files and Class File Format

External representation
(platform independent)

JVM
Internal representation
(implementation dependent)

.load

.classes
.primitive types
.arrays
.strings
.objects
.methods
JVM Data Types

Primitive types:
- numeric integral: byte, short, int, long, char
- numeric floating point: float, double
- boolean: boolean (support only for arrays)
- internal, for exception handling: returnAddress

Reference types:
- class types
- array types
- interface types

Note:
- No type information on local variables at runtime
- Types of operands specified by opcodes (eg: iadd, fadd, ....)
Object Representation

• Left to the implementation
  – Including concrete value of `null`
• This adds extra level of indirection
  – need pointers to instance data and class data
  – make garbage collection easier
• Object representation must include
  – mutex lock
  – GC state (flags)
JVM Runtime Data Areas

**Stack**
- Per Thread Area
  - Thread
  - Program Counter
  - Stack
  - Native Stack

**Non Heap**
- Code Cache
- Permanent Generation
- Interned Strings
- Method Area

**Heap**
- Young Generation
- Old / Tenured Generation
- Eden Space
- Survivor Spaces

Shared among Threads
Threads

• JVM allows multiple threads per application, starting with main
• Created as instances of Thread invoking start() (which invokes run() )
• Several background (daemon) system threads for
  – Garbage collection, finalization
  – Signal dispatching
  – Compilation, etc.
• Threads can be supported by time-slicing and/or multiple processors
Threads (2)

• Threads have shared access to heap and persistent memory
• Complex specification of consistency model
  – volatiles
  – working memory vs. general store
  – non-atomic longs and doubles
• The *Java programming language memory model* prescribes the behaviour of multithreaded programs (JLS Ch. 17)
Java Thread Life Cycle

1.1 Il ciclo di vita di un thread

- Thread.start() / ready
- Object.notify() / Thread.interrupt() / timeout
- sleep finished / Thread.interrupt()
- expired time slot / Thread.yield()
- I/O data ready or monitor free
- return from run() / Thread.stop()
- I/O or monitor request

- ready
- selected
- in execution
- Object.wait() / suspended
- Object.notify() / Thread.interrupt() / timeout
- sleep finished / Thread.interrupt()
- expired time slot / Thread.yield()
- I/O data ready or monitor free
- return from run() / Thread.stop()
- I/O or monitor request

- blocked
- terminated
Per Thread Data Areas

- **pc**: pointer to next instruction in *method area*
  - undefined if current method is *native*
- The **java stack**: a stack of *frames* (or *activation records*).
  - A new frame is created each time a method is invoked and it is destroyed when the method completes.
  - The JVMS does not require that frames are allocated contiguously
- The **native stack**: is used for invocation of native functions, through the JNI (Java Native Interface)
  - When a native function is invoked, eg. a C function, execution continues using the native stack
  - Native functions can call back Java methods, which use the Java stack
Structure of frames

• **Local Variable Array** (32 bits) containing
  – Reference to this (if instance method)
  – Method parameters
  – Local variables

• **Operand Stack** to support evaluation of expressions and evaluation of the method
  – Most JVM bytecodes manipulate the stack

• Reference to **Constant Pool** of current class
Dynamic Linking (1)

• The reference to the constant pool for the current class helps to support **dynamic linking**.

• In C/C++ typically multiple object files are linked together to produce an executable or dll.
  – During the linking phase symbolic references are replaced with an actual memory address relative to the final executable.

• In Java this linking phase is done **dynamically** at runtime.

• When a Java class is compiled, all references to variables and methods **are stored in the class's constant pool as symbolic references**.
Dynamic Linking (2)

• The JVM implementation can choose when to resolve symbolic references.
  – **Eager or static resolution**: when the class file is verified after being loaded
  – **Lazy or late resolution**: when the symbolic reference is used for the first time
• The JVM **has to behave** as if the resolution occurred when each reference is first used and throw any resolution errors at this point.
• **Binding** is the process of the entity (field, method or class) identified by the symbolic reference being replaced by a direct reference
• This only happens once because the symbolic reference **is completely replaced** in the constant pool
• If the symbolic reference refers to a class that has not yet been resolved then this class will be loaded.
Data Areas Shared by Threads: **Heap**

- Memory for objects and arrays; unlike C/C++ they are never allocated to stack
- Explicit deallocation not supported. Only by garbage collection.
- The HotSpot JVM includes four **Generational Garbage Collection Algorithms**
- Since Oracle JDK 11: **Z Garbage Collector**
Data Areas Shared by Threads: **Non-Heap**

- Memory for objects which are never deallocated, needed for the JVM execution
  - Method area
  - Interned strings
  - Code cache for JIT
JIT compilation

- The Hotspot JVM (and other JVMs) profiles the code during interpretation, looking for “hot” areas of byte code that are executed regularly.
- These parts are compiled to native code.
- Such code is then stored in the code cache in non-heap memory.
Method area

The memory where class files are loaded. For each class:

- **ClassLoader Reference**
- From the class file:
  - Run Time Constant Pool
  - Field data
  - Method data
  - Method code

**Note:** Method area is shared among thread. Access to it has to be **thread safe**.

Changes of method area when:

- A new class is loaded
- A symbolic link is resolved by dynamic linking
## Class file structure

```java
ClassFile {
    u4 magic;                        \ 0xCAFEBABE
    u2 minor_version;
    u2 major_version;

    u2 constant_pool_count;
    cp_info  constant_pool[constant_pool_count–1];

    u2 access_flags;\  access modifiers and other info
    u2 this_class;
    u2 super_class;

    u2 interfaces_count;
    u2 interfaces[interfaces_count];

    u2 fields_count;
    field_info  fields[fields_count];\ Static and Instance Variables

    u2 methods_count;
    method_info  methods[methods_count];\ Methods

    u2 attributes_count;
    attribute_info attributes[attributes_count];\ Other Info on the Class
}
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>magic</td>
<td>0xCAFEBABE</td>
</tr>
<tr>
<td>minor_version</td>
<td>Java Language Version</td>
</tr>
<tr>
<td>major_version</td>
<td>Constant Pool</td>
</tr>
<tr>
<td>constant_pool_count</td>
<td>Constant Pool</td>
</tr>
<tr>
<td>access_flags</td>
<td>access modifiers and other info</td>
</tr>
<tr>
<td>this_class</td>
<td>References to Class and Superclass</td>
</tr>
<tr>
<td>super_class</td>
<td>References to Class and Superclass</td>
</tr>
<tr>
<td>interfaces_count</td>
<td>References to Direct Interfaces</td>
</tr>
<tr>
<td>fields_count</td>
<td>Static and Instance Variables</td>
</tr>
<tr>
<td>methods_count</td>
<td>Methods</td>
</tr>
<tr>
<td>attributes_count</td>
<td>Other Info on the Class</td>
</tr>
</tbody>
</table>
Field data in the Method Area

Per field:

• Name
• Type
• Modifiers
• Attributes
# FieldType descriptors

<table>
<thead>
<tr>
<th>FieldType term</th>
<th>Type</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>byte</td>
<td>signed byte</td>
</tr>
<tr>
<td>C</td>
<td>char</td>
<td>Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16</td>
</tr>
<tr>
<td>D</td>
<td>double</td>
<td>double-precision floating-point value</td>
</tr>
<tr>
<td>F</td>
<td>float</td>
<td>single-precision floating-point value</td>
</tr>
<tr>
<td>I</td>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>J</td>
<td>long</td>
<td>long integer</td>
</tr>
<tr>
<td>L ClassName ;</td>
<td>reference</td>
<td>an instance of class ClassName</td>
</tr>
<tr>
<td>S</td>
<td>short</td>
<td>signed short</td>
</tr>
<tr>
<td>Z</td>
<td>boolean</td>
<td>true or false</td>
</tr>
<tr>
<td>[</td>
<td>reference</td>
<td>one array dimension</td>
</tr>
</tbody>
</table>
Method data

Per method:
• Name
• Return Type
• Parameter Types (in order)
• Modifiers
• Attributes
• Method code...

A method descriptor contains
• a sequence of zero or more parameter descriptors in brackets
• a return descriptor or V for void descriptor

Example: The descriptor of

Object m(int i, double d, Thread t) {...}

is:

(IDLjava/lang/Thread;)Ljava/lang/Object;
Method code

Per method:
• Bytecodes
• Operand stack size
• Local variable size
• Local variable table
• Exception table
• LineNumberTable – which line of source code corresponds to which byte code instruction (for debugger)

Per exception handler (one for each try/catch/finally clause)
• Start point
• End point
• PC offset for handler code
• Constant pool index for exception class being caught
Disassembling Java files: javac, javap, java

```java
package org.jvminternals;
public class SimpleClass {
    public void sayHello() {
        System.out.println("Hello");
    }
}
```

Compiler
```
javac SimpleClass.java
```

Disassembler
```
javap -c -v SimpleClass.class
```

JVM
```
java SimpleClass
```
SimpleClass.class: constructor and method

Local variable 0 = “this”

package org.jvminternals;
public class SimpleClass {
    public org.jvminternals.SimpleClass();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
        stack=1, locals=1, args_size=1
        0: aload_0
        1: invokespecial #1    // Method java/lang/Object."<init>":()V
        4: return
    LineNumberTable:
        line 2: 0

    public void sayHello();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
        stack=2, locals=1, args_size=1
        0: getstatic #2    // Field java/lang/System.out:Ljava/io/PrintStream;
        3: ldc #3    // String Hello
        5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        8: return
    LineNumberTable:
        line 4: 0
        line 5: 8
}
SourceFile: "SimpleClass.java"
The constant pool

• Similar to symbol table, but with more info
• Contains constants and symbolic references used for dynamic binding, suitably tagged
  – numeric literals (Integer, Float, Long, Double)
  – string literals (Utf8)
  – class references (Class)
  – field references (Fieldref)
  – method references (Mehodref, InterfaceMethodref, MethodHandle)
  – signatures (NameAndType)
• Operands in bytecodes often are indexes in the constant pool
public class SimpleClass {
    public void sayHello() {
        System.out.println("Hello");
    }
}

SimpleClass.class: the Constant pool
Compiled from "SimpleClass.java"
public class SimpleClass
    minor version: 0
    major version: 52
    flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
    #1 = Methodref          #6.#14  // java/lang/Object."<init>":()V
    #2 = Fieldref           #15.#16 // java/lang/System.out:Ljava/io/PrintStream;
    #3 = String             #17    // Hello
    #4 = Methodref           #18.#19 // java/io/PrintStream.println:(Ljava/lang/String;)V
    #5 = Class               #20    // SimpleClass
    #6 = Class               #21    // java/lang/Object
    #7 = Utf8                <init>
    #8 = Utf8                ()V
    #9 = Utf8                Code
    #10 = Utf8               LineNumberTable
    #11 = Utf8               sayHello
    #12 = Utf8               SourceFile
    #13 = Utf8               SimpleClass.java
    #14 = NameAndType        #7:#8   // "<init>":()V
    #15 = Class              #22    // java/lang/System
    #16 = NameAndType        #23:#24 // out:Ljava/io/PrintStream;
    #17 = Utf8               Hello
    #18 = Class              #25    // java/io/PrintStream
    #19 = NameAndType        #26:#27 // println:(Ljava/lang/String;)V
    #20 = Utf8               SimpleClass
    #21 = Utf8               java/lang/Object
    #22 = Utf8               java/lang/System
    #23 = Utf8               out
    #24 = Utf8               Ljava/io/PrintStream;
    #25 = Utf8               java/io/PrintStream
    #26 = Utf8               println
    #27 = Utf8               (Ljava/lang/String;)V

public void sayHello();
    descriptor: ()V
    Code:
        stack=2, locals=1, args_size=1
        0: getstatic     #2
        3: ldc          #3
        5: invokevirtual #4
        8: return
public void sayHello();

descriptor: ()V

Code:
stack=2, locals=1, args_size=1
0: getstatic #2
3: ldc #3
5: invokevirtual #4
8: return
Loading, Linking, and Initializing

• **Loading**: finding the binary representation of a class or interface type with a given name and creating a class or interface from it

• **Linking**: taking a class or interface and combining it into the run-time state of the Java Virtual Machine so that it can be executed

• **Initialization**: executing the class or interface initialization method `<clinit>`
JVM Startup

• The JVM starts up by loading an initial class using the **bootstrap classloader**
• The class is linked and initialized
• `public static void main(String[])` is invoked.
• This will trigger loading, linking and initialization of additional classes and interfaces...
Loading

• Class or Interface $C$ creation is triggered
  – by other class or interface referencing $C$
  – by certain methods (eg. reflection)
• Array classes are generated by the JVM
• Check whether already loaded
• If not, invoke the appropriate loader.loadClass
• Each class is tagged with the \textit{initiating loader}
• \textit{Loading constraints} are checked during loading
  – to ensure that the same name denotes the same type in different loaders
Class Loader Hierarchy

- **Bootstrap Classloader** loads basic Java APIs, including for example `rt.jar`. It may skip much of the validation that gets done for normal classes.

- **Extension Classloader** loads classes from standard Java extension APIs such as security extension functions.

- **System Classloader** is the default application classloader, which loads application classes from the classpath.

- **User Defined Classloaders** can be used to load application classes:
  - for runtime reloading of classes
  - for loading from different sources, e.g. from network, from an encrypted file, or also generated on the fly
  - for supporting separation between different groups of loaded classes as required by web servers

- Class loader hooks: `findClass` (builds a byte array), `defineClass` (turns an array of bytes into a class object), `resolveClass` (links a class)
Runtime Constant Pool

• The constant_pool table in the .class file is used to construct the run-time constant pool upon class or interface creation.

• All references in the run-time constant pool are initially symbolic.

• Symbolic references are derived from the .class file in the expected way.

• Class names are those returned by `Class.getName()`.

• Field and method references are made of name, descriptor and class name.
Linking

• Link = verification, preparation, resolution
• **Verification**: see below
• **Preparation**: allocation of storage (method tables)
• **Resolution** (optional): resolve symbol references by loading referred classes/interfaces
  – Otherwise postponed till first use by an instruction
Verification

• When?
  – Mainly during the load and link process

• Why?
  – No guarantee that the class file was generated by a Java compiler
  – Enhance runtime performance

• Examples
  – There are no operand stack overflows or underflows.
  – All local variable uses and stores are valid.
  – The arguments to all the JVM instructions are of valid types.

• Relevant part of the JVM specification: described in ~170 pages of the JVMS (total: ~600 pages)
Verification Process

• Pass 1 – when the class file is loaded
  – The file is properly formatted, and all its data is recognized by the JVM

• Pass 2 – when the class file is linked
  – All checks that do not involve instructions
    • final classes are not subclassed, final methods are not overridden.
    • Every class (except `Object`) has a superclass.
    • All field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.
Verification Process – cont.

- Pass 3 – still during linking
  - **Data-flow analysis** on each method.
  - Ensure that at any given point in the program, no matter what code path is taken to reach that point:
    - The operand stack is always the same size and contains the same types of objects.
    - No local variable is accessed unless it is known to contain a value of an appropriate type.
    - Methods are invoked with the appropriate arguments.
    - Fields are assigned only using values of appropriate types.
    - All opcodes have appropriate type arguments on the operand stack and in the local variables.
    - A method must not throw more exceptions than it admits.
    - A method must end with a return value or throw instruction.
    - Method must not use one half of a two word value.
Verification Process – cont.

- Pass 4 - the first time a method is actually invoked
  – a virtual pass whose checking is done by JVM instructions
    - The referenced method or field exists in the given class.
    - The currently executing method has access to the referenced method or field.
Initialization

• <clinit> initialization method is invoked on classes and interfaces to initialize class variables
• static initializers are executed
• direct superclass need to be initialized prior
• happens on direct use: method invocation, construction, field access
• synchronized initializations: state in Class object
• <init>: initialization method for instances
  – invokespecial instruction
  – can be invoked only on uninitialized instances
Initialization example (1)

class Super {
    static {
        System.out.print("Super ");
    }
}
class One {
    static {
        System.out.print("One ");
    }
}
class Two extends Super {
    static {
        System.out.print("Two ");
    }
}
class Test {
    public static void main(String[] args) {
        One o = null;
        Two t = new Two();
        System.out.println((Object)o == (Object)t);
    }
}

What doers java Test print?

Super  Two  False
Initialization example (2)

class Super { static int taxi = 1729; }
}
class Sub extends Super {
    static { System.out.print("Sub "); }
}
class Test {
    public static void main(String[] args) {
        System.out.println(Sub.taxi);
    }
}

What does \texttt{java Test} print?

Only prints "1729"

A reference to a static field (§8.3.1.1) causes initialization of only the class or interface that actually declares it, even though it might be referred to through the name of a subclass, a subinterface, or a class that implements an interface. (page 385 of [JLS-8])
Finalization

• Invoked just before garbage collection
• JLS does not specify when it is invoked
• Also does not specify which thread
• No automatic invocation of super’s finalizers
• Very tricky!

```java
void finalize() {
    classVariable = this; // the object is reachable again
}
```

• Each object can be
  – Reachable, finalizer-reachable, unreachable
  – Unfinalized, finalizable, finalized
**Finalization State Diagram**

https://notendur.hi.is/snorri/SDK-docs/lang/lang083.htm

**finalize()** is never called a second time on the same object, but it can be invoked as any other method!
JVM Exit

- `classFinalize` similar to object finalization
- A class can be unloaded when
  - no instances exist
  - class object is unreachable
- JVM exits when:
  - all its non-daemon threads terminate
  - `Runtime.exit` or `System.exit` assuming it is secure
- finalizers can be optionally invoked on all objects just before exit
Resources

• JVMS Chapter 2 - The Structure of the Java Virtual Machine

• *JVM Internals, by James D. Bloom*
  
  [http://blog.jamesdbloom.com/JVMInternals.html](http://blog.jamesdbloom.com/JVMInternals.html)

• JLS Chapter 17 – Memory model