301AA - Advanced Programming

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

"HOW" THE PROGRAM IS EXECUTED
- TURING MACHINES
- LAMDA CALCULUS
- ....
- != PARADIGM

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

TO THE ABSTRACT MACHINE

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

IN JAVA FOR EXAMPLE THE GARBAGE COLLECTOR

EXTERNAL FUNCTIONALITIES OF OUR AB. MACHINE

SYSTEM CALLS ARE PART OF THE RUNTIME SYSTEM
- I/O CALLS

PART OF THE RUNTIME SYS. SINCE THEY IMPLEMENT THE EXECUTION MODEL

Runtime Support needed for…

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

- 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.

ALL THOSE DATA NEEDED BY THE PROGRAM IN EXECUTION BUT NOT PART OF THE PROGRAM ITSELF (OS BUFFERS, …)

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

COMPILATION DURING EXECUTION

OF PROPERTIES THAT WE WANT TO HAVE DURING THE EXECUTION (EX. NO MEM. LEAK) REQUIRES RS AND MONITORING

TO DEBUG YOU NEED THE CURRENT VALUES OF VARS, BREAKPOINTS AND OTHER INFO AVAILABLE ONLY AT RUNTIME => NEED RS

Java Runtime Enviroment - JRE

- Includes all what is needed to run compiled Java programs (COMPILED JAVA IS BYTECODE)
  - JVM – Java Virtual Machine (THAT HAS AS MACHINE LANGUAGE THE BYTECODE)
  - JCL – Java Class Library (Java API) (JAVA UTILS FOR EXAMPLE)
- We shall focus on the JVM as a real runtime system covering most of the functionalities just listed

JVM UPDATES

- SIDE NOTE: JAVA VERSIONS CHANGE PRETTY OFTEN (WAY MORE THAN USED TO BE) BUT THEY MUST GUARANTEE RETROCOMPATIBILITY
  - EVERYTHING THAT WAS RUNNING ON THE OLD VERSION HAS TO RUN TO THE NEW VERSION
    - NOT EVERYONE DO THIS (EX: PYTHON 2 VS 3)

IMPOSE (ABSTRACT) PROPERTIES THAT HAVE TO BE RESPECTED, TRYING TO GIVE ZERO "HINTS" ABOUT THE IMPLEMENTATION

What is the JVM?

ORACLE DON'T IMPOSE AN IMPLEMENTATION OF 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

ALLOWS STATIC ANALYSIS, SEE LATER ON

THE JVM IS NOT PORTABLE! BY NOT GIVING RULES OF IMPLEMENTATION BUT ONLY CONCEPTUAL PROPERTIES/CONSTRAINTS ANYONE CAN IMPLEMENT THE JVM TAILORED FOR HIS ARCHITECTURE

MORE THAN ONE FLOW OF EXECUTION GOING ON AT THE SAME TIME
- THREADS SHARE MEMORY
- GLOBAL, HEAP AND CODE

Execution model

NOT RELATED TO THE CALL STACK

- 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

WE WILL SEE LATER ON ☺

STACK BASED ARCHITECTURE: THE OP OF THE JVM OPERATES ONLY ON THE VALUES THAT ARE CONTAINED IN THE STACK. DATA HAVE TO BE BROUGHT IN THE STACK TO BE ABLE TO USE IT

→ KINDA REGISTER MACHINE BUT WITH THE OPERANDS STACK

- OPERATION READ/POP OPERANDS ON THE STACK AND PUSH ON TOP THE RESULT

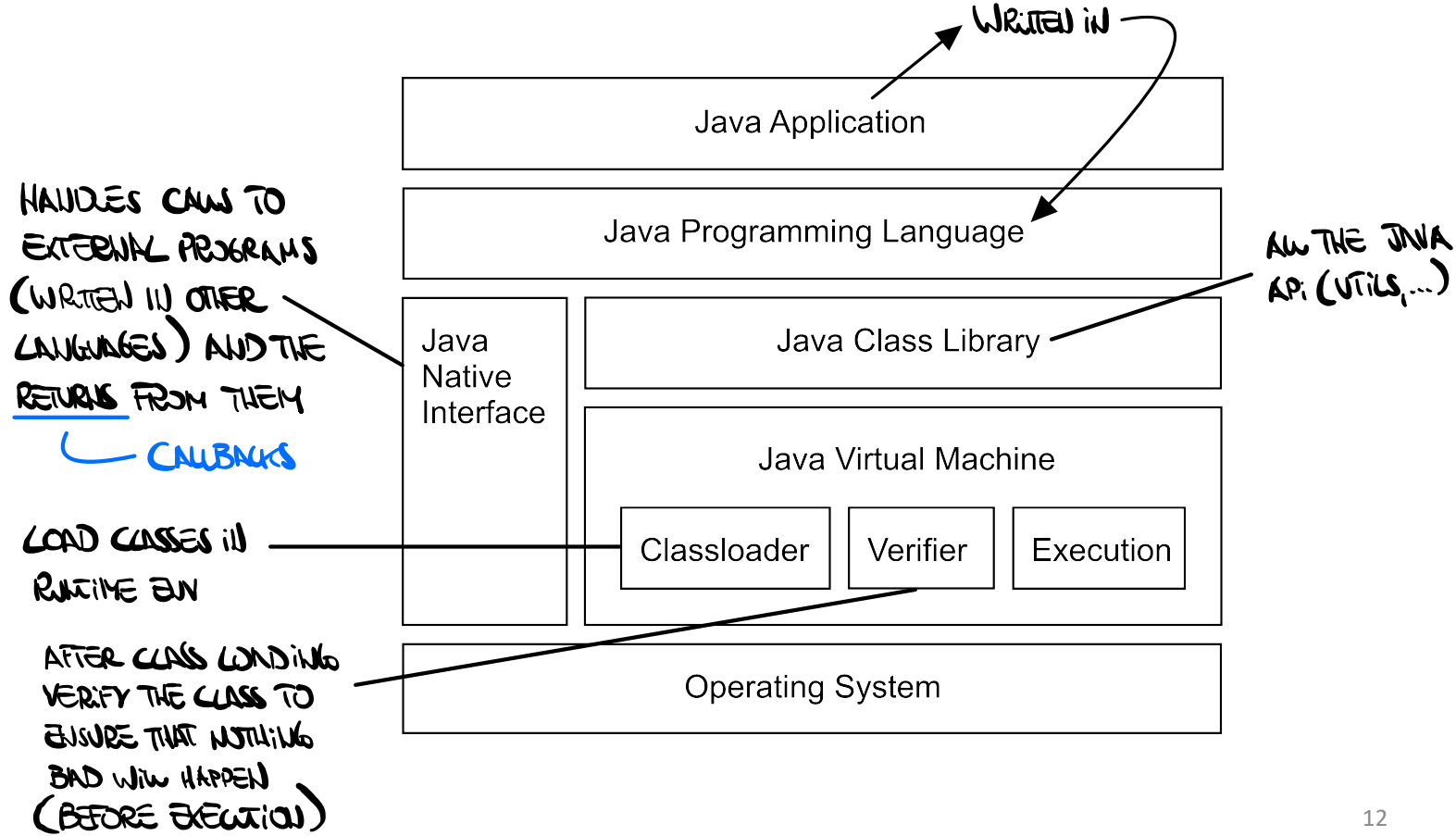
THE OPERAND STACK IS INSIDE THE ACTIVATION RECORD

CALL STACK

- EACH FUNCTION GENERATES A PUSH OF THE AR ON THE STACK AND INSIDE EACH AR THERE IS AN OPERAND STACK ON WHICH THE JVM DO THE STUFF OF THE FUNCTION

* HINTS BUT LATER

Java Abstact Machine Hierarchy



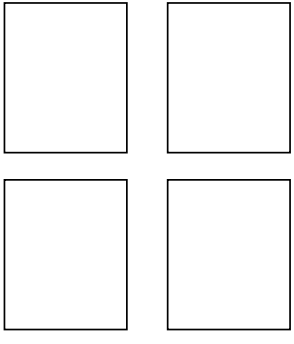
Class Files and Class File Format

THE .CLASS FILES HAVE A PRECISE SYNTAX WHICH IS INDEPENDENT FROM THE ARCHITECTURE AND THE IMPLEMENTATION OF THE JVM

HOW STUFF IS IN MEMORY AND SO ON

External representation
(platform independent)

.class files



load

JVM

Internal representation
(implementation dependent)

classes

primitive types

arrays

objects

strings

methods

THE LOADING PROCESS TRANSFORMS .CLASS FILES TO AN INTERNAL REPRESENTATION THAT IS SPECIFIC FOR THIS IMPLEMENTATION OF THE JVM

- THE LOADING PROCESS MUST BE IMPLEMENTED WITH THE IMPLEMENTATION OF THE JVM

COMPILED JAVA CODE

JAVAC FILE.JAVA → FILE.CLASS

this is bytecode!

JVM Data Types

DIMENSION DEFINED BY THE JVM SPEC, NOT IMPLEMENTATION DEPENDANT!

UNSIGNED SHORT, 2 BYTES AS UNICODE (NOW NOT ASCII)

1 BYTE 2 BYTES 4 BYTES 8 BYTES

Primitive types:

- numeric integral: `byte`, `short`, `int`, `long`, `char` ⊛
- numeric floating point: `float`, `double`, RESPECTIVELY 4 AND 8 BYTES
- boolean: `boolean` (support only for arrays) ⊛
- internal, for exception handling: `returnAddress` } CAN'T BE USED BY THE USER (DEV)

Reference types: (POINTER (TO HEAP) TYPES: ALWAYS 4 BYTES)

- class types
- array types
- interface types

Note:

ADD ON INTEGERS

ADD ON FLOATS

- No type information on local variables at runtime ⊛
- Types of operands specified by **opcodes** (eg: `iadd`, `fadd`, ….)
- STRINGS ARE IMMUTABLE OBJECTS
- ARRAYS ARE OBJECTS EVEN IF THEY HAVE A TYPE-LIKE SYNTAX. SUPERCLASS OBJECT AND ALLOCATED ON THE HEAP

- A LOT OF OPERATIONS ARE SUPPORTED ONLY ON INT, FLOAT, DOUBLE
  - TO SUM [2] BYTES I NEED TO CONVERT THEM TO INTEGERS
    THEN USE "IADD" AND CONVERT BACK THE RESULT

- BOOLEAN IS PRIMITIVE FOR THE LANGUAGE BUT NOT FOR THE JVM!
  OPERATIONS ARE NOT FULLY SUPPORTED

THIS IS BECAUSE THE SMALLEST INDIVIDUABLE WORD IS OF 1 BYTE HENCE WE CAN'T HAVE OPERATIONS ON BOOL THAT TAKES ONLY 1 BIT (CAN'T READ A BIT, AT LEAST 1 BYTE)

- BOOLEAN ARRAYS (>8 ELEMENTS) ARE SUPPORTED SINCE ALLOCATED CONTIGUOSLY
- TO SAVE A BIT (BOOLEAN) WE TAKE A FULL BYTE :C

- JVM OPERATIONS ARE TYPED AND THE RIGHT OPERATION IS CHOSEN USING THE TYPE ANNOTATION GIVEN BY THE USER (DEVELOPER)

Object Representation
IN THE MEMORY

- Left to the implementation (NOT DEFINED IN THE SPECIFICATION)
  - Including concrete value of `null`
- 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)

THE ACTUAL VALUE OF NULL IS NOT SPECIFIED, IT DEPENDS ON THE IMPLEMENTATION
- MOST OFTEN IS ZERO
- NOT GOOD CHOICE, WE'LL SEE

IMPLEMENTATION DEPENDENT

ALSO ALLOWS US TO DO REFERENCE COUNTING FOR GARBAGE COLLECTION

SINCE OBJECT STRUCTURE IS NOT DEFINED (IMPLEM. DEPEND.) WE CAN ALLOCATE OBJECTS ON THE STACK. ON THE STACK WE HAVE POINTERS (ALWAYS 4 BYTES) TO HEAP

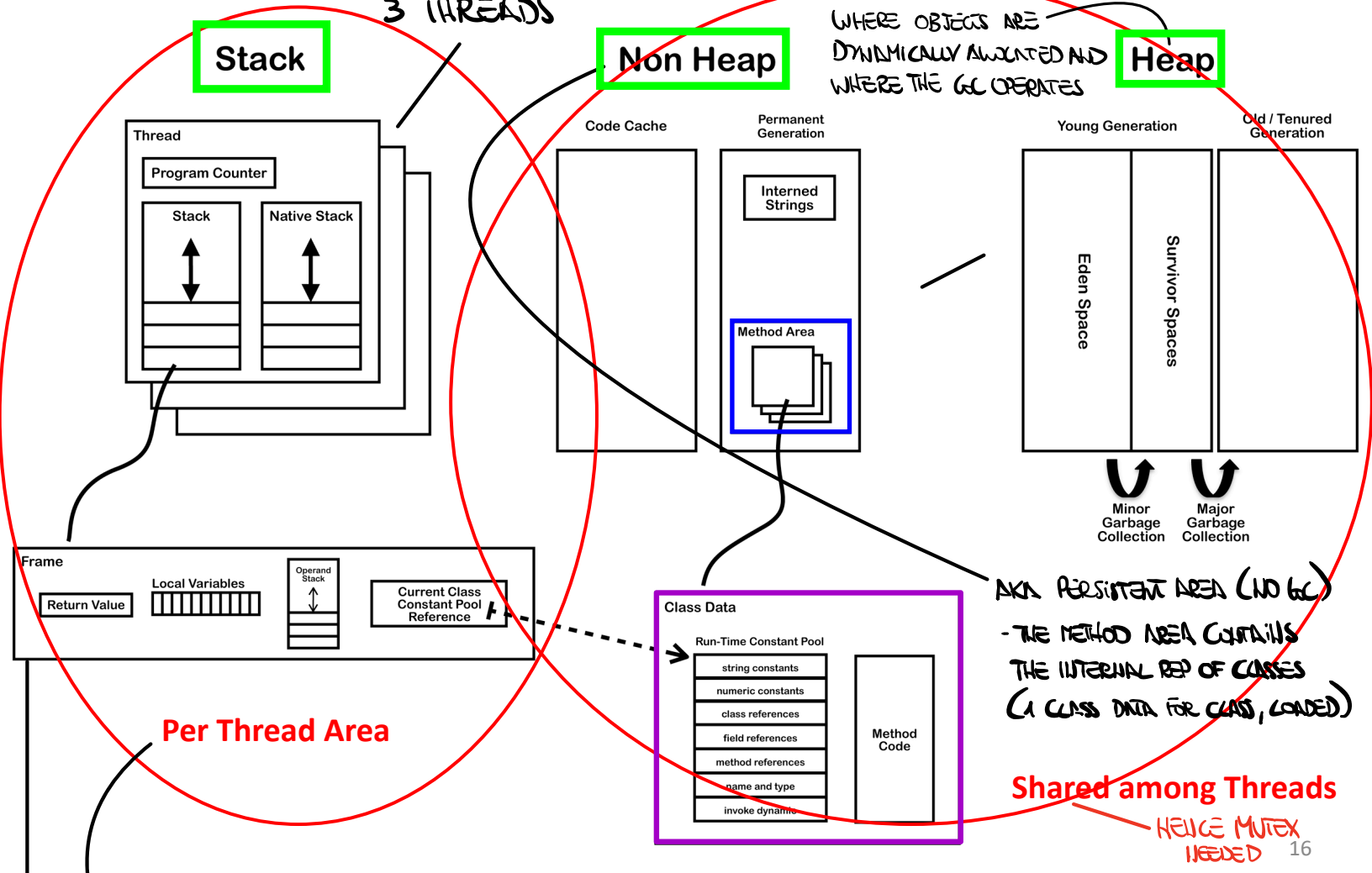
- JVM IS CONCURRENT AND WITH GARBAGE COLLECTION THE JVM SPECIFIC IMPLICIT THAT HAVE TO BE PRESENT MUTEX TO GUARANTEE MUTUAL EXCLUSION (IS THIS OBJECT LOCKED? WHICH THREAD HAS THE LOCK?)

JVM Runtime Data Areas

THE FOLLOWING STUFF IS NEEDED IN THE JVM RDA SPECIFICATION

3 THREADS

WHERE OBJECTS ARE DYNAMICALLY ALLOCATED AND WHERE THE GC OPERATES



AKA PERSISTENT AREA (NO GC)

- THE METHOD AREA CONTAINS THE INTERNAL REP OF CLASSES (1 CLASS DATA FOR CLASS, LOADED)

Shared among Threads

HENCE MUTEX NEEDED

ONE THREAD AREA FOR EACH THREAD IN EXECUTION

- EACH THREAD HAS ITS OWN PROGRAM COUNTER THAT POINTS TO THE ADDRESS OF THE NEXT (BYTECODE) INSTRUCTION OF THE METHOD IN "EXECUTION" (WHICH IS FOUND BY THE CURRENT CLASS CONSTANT POOL REFERENCE), ITS OWN STACK OF AR TO HANDLE FUNCTIONS AND ITS NATIVE STACK, TO EXECUTE NATIVE CODE

IN EACH ACTIVATION RECORD WE HAVE

- OPERAND STACK: THE STACK USED BY THE STACK MACHINE (JVM) TO PERFORM THE BODY OF THE FUNCTION
- ARRAY OF LOCAL VARIABLES: THE LOCAL VARS USED BY THE FUNCTION RELATIVE TO THE AR
- RETURN VALUE: IT MAYBE COPIED LATER ON BY THE JVM IF IT'S NEEDED TO ASSIGN THE VALUE TO SOMETHING (X = OBJ.METHOD())
- CURRENT CLASS CONSTANT POOL REFERENCE: POINTS TO THE CLASS DATA THAT CONTAINS THE METHOD THAT WE ARE EXECUTING

- CLASS DATA CONTAINS THE RUNTIME CONSTANT POOL (NEED FOR THE DYNAMIC BINDING BETWEEN FUNCTION AND IMPLEMENTATION) AND THE ACTUAL BYTECODE OF CLASS'S METHODS, TYPE OF PARAMS AND TYPE OF RETURN

JVM IS MULTITHREADED AND HAS MULTIPLE THREADS IN EXECUTION FOR THE RUNTIME SUPPORT
- EVEN IF THE "USER" CODE IS SINGLE THREAD THERE ARE ALWAYS MORE THREADS RUNNING

Threads

- JVM allows multiple threads per application, starting with `main` ——— 1ST THREAD STARTED
- Created as instances of `Thread` invoking `start()` (which invokes `run()`)
- Several background (daemon) system threads for
  - Garbage collection, finalization
  - Signal dispatching
  - Compilation, etc.

NOT PART OF THE APPLICATION LIFE CYCLE
- AN APP. CAN TERMINATE IFF ALL NON-DEAMON THREADS ARE TERMINATED

- Threads can be supported by time-slicing and/or multiple processors (DEPENDS ON HW ARCHITECTURE AND UNDERLYING OS)

START DO THE INTERNAL ALLOCATION AND THEN INVOKE RUN, WHICH IS THE FUNCTION (PASSED AS PARAM)
- THE THREAD WILL START A PARALLEL EXECUTION OF ITS CODE (RUN METHOD CODE)

Threads (2)

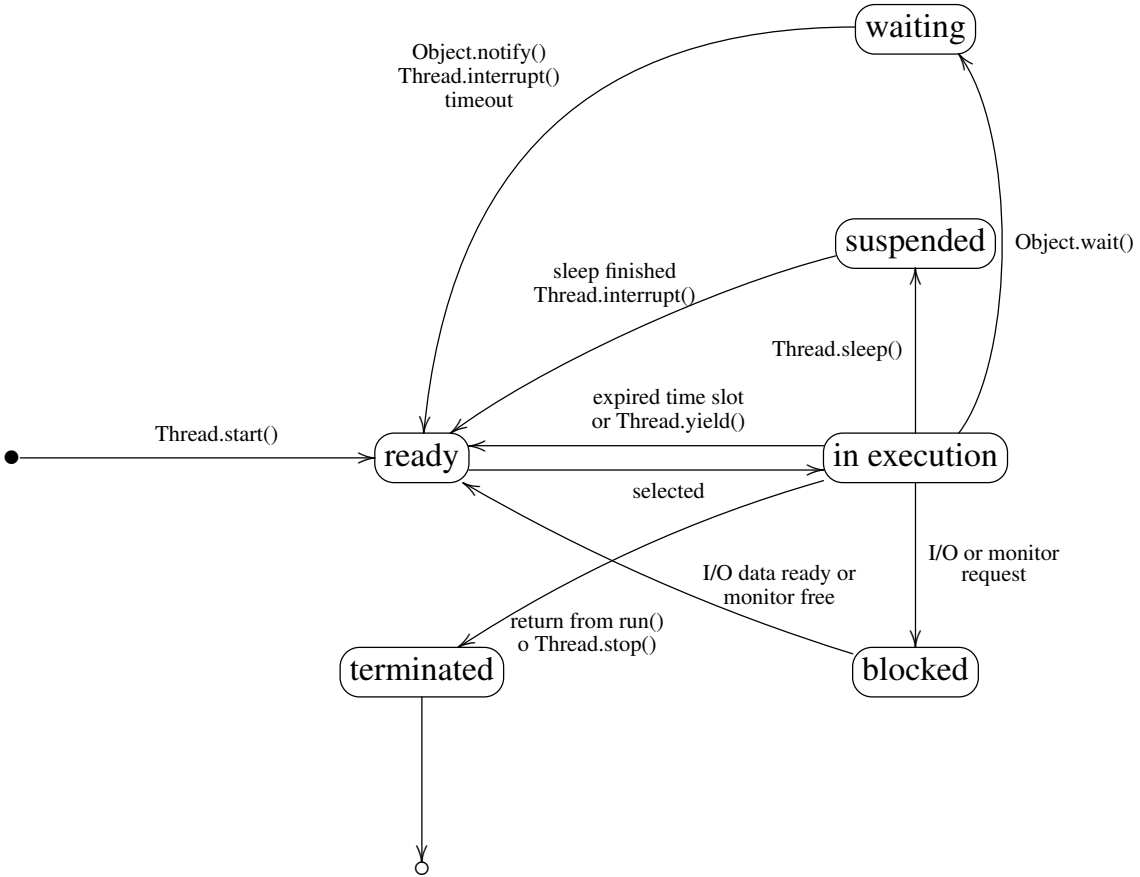
- Threads have shared access to heap and persistent memory
- Complex specification of consistency model
  - volatiles
  - working memory vs. general store
  - non-atomic longs and doubles

SPECIFY WHAT CAN HAPPEN WHEN THREAD CONCURRENTLY ACCESS A RESOURCE

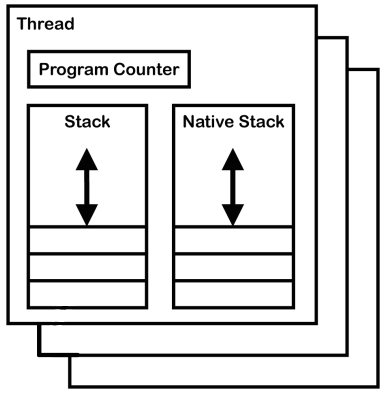
- THE SPECIFICATION OF THE CONSISTENCY MODEL HAS TO BE AS GENERAL AS POSSIBLE, NOT IMPOSING CONSTRAINTS ON IMPLEMENTATION
  - EX: CAN'T IMPOSE 64 BIT ATOMIC READ AS MOST MACHINE WOULDN'T SUPPORT IT

EXAMPLE: THE READ OF 32 BIT IS ATOMIC, THE READ OF 64 BIT IS DONE WITH 2 ATOMIC READS: WHAT HAPPENS IF SOMEONE DO THINGS BETWEEN READS?

Java Thread Life Cycle



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

PARAMETER & RETURN PASSING HAS TO BE HANDLED
• EVEN REPRESENTATION OF DATA

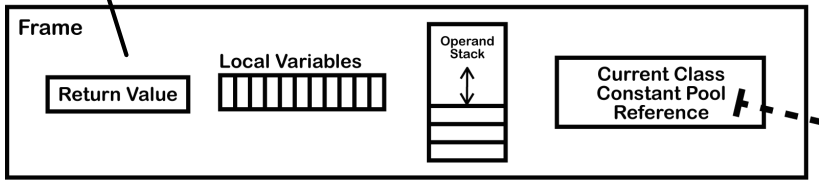
WE REPRESENT IT CONTIGUOUSLY BUT THAT IS NOT IMPOSED: COULD BE IMPLEMENTED AS A LINKED LIST (THE JVM IS AN ABSTRACT MACHINE)

- PC POINTS TO THE NEXT LINE OF CODE TO EXECUTE, HOW WE FIND IT?
  - IN THE ("CURRENT") AR ON THE STACK WE HAVE THE POINTER TO THE CLASS DATA (THE POINTER IS ON THE METHOD AREA)
  - THE CLASS DATA CONTAINS THE METHOD CODE (THE ACTUAL BYTECODE)
  - BY SPECIFICATION IF THE CODE IN EXECUTION IS NOT BYTECODE (EX: C THROUGH JNI) THE JVM DO NOT EVEN KNOW THE EXECUTION MODEL OF THAT (NATIVE) CODE => PC UNDEFINED

- (SEE BELOW) THE SIZE OF BOTH IS NOT FIXED BUT ESTIMATE AT COMPILE TIME (WHEN THE BYTECODE IS GENERATED): VERY EFFICIENT BECAUSE FOR SMALL METHODS WE DO NOT WASTE MEMORY

VALUE OF RETURN OF THE FUNCTION (IF ANY). AT THE DESTRUCTION OF THE AR (WHEN THE FUNCTION RETURNS) THE VALUE IS COPIED ON THE LOCAL VARS ARRAY OF THE CALLER (IF THE RETURN VALUE IS ASSIGNED TO SOMETHING)

Structure of frames



EACH CELL

- **Local Variable Array** (32 bits) containing
  - Reference to `this` (if instance method) — IN POS 0 OF THE ARRAY
  - Method parameters
  - Local variables
- **Operand Stack** to support evaluation of expressions and evalutation of the method
  - Most JVM bytecodes manipulate the stack
- Reference to **Constant Pool** of current class
  - EACH METHOD IS DEFINED IN SOME CLASS (STATIC OR INSTANCE OF..):
    - THIS POINTS TO THE (INTERNAL) REP OF THE CURRENT CLASS IN CONST. POOL THAT CONTAINS THE CODE (BYTECODE)

SUBSTITUTION SIMBOLIC ADDRESSES OF ALREADY COMPILED CODE WITH ACTUAL PHYSICAL ADDRESSES THAN CAN BE USED IN THE EXECUTION

Dynamic Linking (1)

MIGHT BE API, LIBRERES...

- 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**.

IN THE BYTECODE THERE ARE LOGIC ADDRESSES!
THEY ARE RESOLVED ONLY A RUNTIME DURING THE EXECUTION

Dynamic Linking (2)

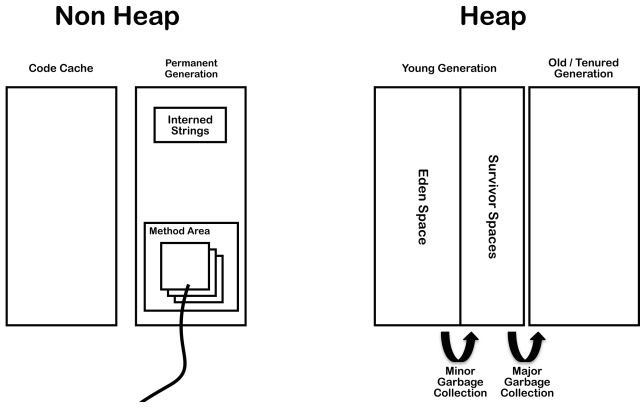
- The JVM implementation can choose when to resolve symbolic references.
  - **Eager or static resolution:** when the class file is verified after being loaded
  - **Lazy or late resolution:** when the symbolic reference is used for the first time
- The JVM **has to behave** as if the resolution occurred 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.

IN THE METHOD AREA AND THEN RESOLVED AFTER THE 1ST TRANSLATION I REPLACE THE OTHER OCCURENCES OF THE LOGIC ADDRESS

- LOADING PHASE THEN VERIFICATION PHASE
  - IN EAGER RESOLUTION IF IN THE VERFICATION PHASE IS FOUND AN UNRESOLVED REFERENCE WE RESOLVE IT
    - THE CLASS THAT CONTAIN THE DATA (EG. METHOD ATTRIBUTE) AND THE LOGIC ADDRESS GET TRANSLATED
      - EAGER: WE RESOLVE ADDRESSES THAT MAY BE NEVER USED IN THE EXECUTION (AN IF BRANCH..)
    - THE EAGER RESOLUTION MAKES THE CLASS LOADING A STATIC EVENT (DONE STATICALLY, NOT AT RUNTIME)
      - THE EXECUTION WILL BE FASTER BUT ERRORS MAYBE SIGNALATED EVEN IF THEY WOULD NOT OCCUR IN THE EXECUTION
- GETS RESOLVED THE FIRST TIME USED AND STORED (NEVER TRANSLATE ADDRESSES TWICE)
- THE SPECIFICATION STATES THAT THE JVM HAS TO BEHAVE AS THE RESOLUTION IS DONE WHEN THE REFERENCE IS FIRST USED (AND THROW ERRORS IF ANY)

  → IF YOU USE AN EAGER APPROACH YOU MAY ALREADY KNOW FROM THE LOADING PHASE THAT THERE IS A PROBLEM, BUT YOU CAN'T REPORT IT UNTIL THE REFERENCE IS USED

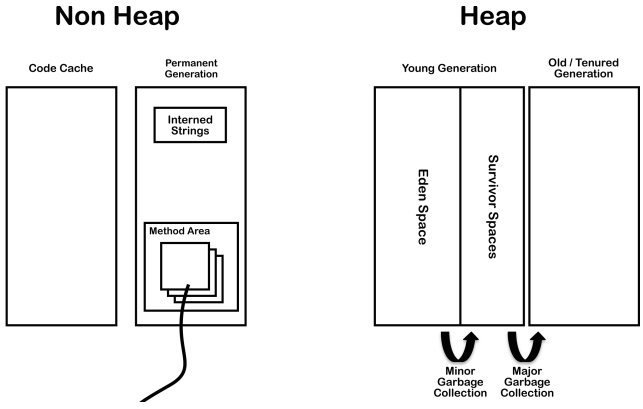
Data Areas Shared by Threads: **Heap**



- Memory for objects and arrays; unlike C/C++ they are never allocated to stack
- Explicit deallocation not supported. Only by garbage collection.

- 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 — HERE THE INTERNAL REP OF CLASSES
  - Interned strings — EXPLICIT STRINGS BETWEEN "". STILL OBJECTS AND THEY ARE SAVED HERE SINCE THEY ARE NOT ALLOCATED MULTIPLE TIMES - STRING IS FINAL AND STRINGS ARE UNMUTABLE
  - Code cache for JIT

GENERALLY IN THE EXECUTION OF A PROGRAM THERE IS A PART OF THE MEMORY THAT IS STATIC (LIKE THE CODE) THAT IS ALLOCATED WHEN THE PROGRAM STARTS AND DEALLOCATED WHEN THE PROGRAM ENDS. THEN THERE'S THE DINAMIC PART OF THE MEMORY LIKE THE HEAP THAT GROWS AND SHRINKS DURING THE EXECUTION

- THIS NON-HEAP IS SEMI STATIC: NOT FILLED COMPLETELY AT THE START (METHOD AREA FOR EXAMPLE MAY GROW) BUT IS ALMOST NEVER AFFECTED BY THE GC

JIT (Just In Time) compilation

ORACLE'S IMPLEMENTATION OF THE JVM

- 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.
- EFFICIENCY BOOST IN EXECUTION TIME

Method area

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

- **Classloader Reference**
- From the `class` file:
  - **Run Time Constant Pool**
  - **Field data** (N, scoping, static, ..)
  - **Method data**
  - **Method code** (BYTECODE)

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

WE WILL SEE IT LATER ON

REMEMBER: .CLASS FILES ARE IMPLEM. INDIPENDENT! DURING THE LOADING PHASE THERE IS A PASSAGE FROM MACHINE INDIP. TO INTERNAL REPRESENTATION

IN THE .CLASS FILE THERE IS THE (STATIC) CONSTANT POOL, THAT IN THE LOADING IS TRANSFORMED "1-TO-1" IN THE RT CONST P

THE RESOLUTION IS DONE BY REPLACING THE LOGIC ADDRESSES WITH THE PHISICAL ADDRESSES IN THE RUNTIME CONSTANT POOL
- HENCE THIS PROCESS ALTERS THE METHOD AREA

FOR EACH METHOD:
#ARGUMENTS, ARGS TYPES, RETURN TYPE

Class file structure
(BY JVM SPECIFICATION)

uN: UNSIGNED N BYTES

ClassFile {

u4	magic;	0xCAFEBABE
u2	minor_version;	Java Language Version
u2	major_version;	
u2	constant_pool_count;	Constant Pool
cp_info	contant_pool[constant_pool_count–1];	
u2	access_flags;	access modifiers and other info
u2	this_class;	References to Class and Superclass
u2	super_class;	
u2	interfaces_count;	References to Direct Interfaces
u2	interfaces[interfaces_count];	
u2	fields_count;	Static and Instance Variables
field_info	fields[fields_count];	
u2	methods_count;	Methods
method_info	methods[methods_count];	
u2	attributes_count;	Other Info on the Class
attribute_info	attributes[attributes_count];	

}

INDICATES THAT THIS IS A JAVA CLASSFILE

#ELEMENTS IN STATIC CONST POOL

ACTUAL CONST POOL

(PRIVATE, PUBLIC, PROTECTED, ...)

#INTERFACES OF THIS CLASS

HOST 1 LEVEL UP

#FIELDS

ACTUAL FIELDS

FIELDS ACCESS (PRIVATE, ...)

#METHODS

METHODS CODES (BYTECODE)

METHODS ACCESS (PRIVATE, STATIC, ...), SIGNATURES, #NAMES & TYPES, RETURN TYPE

DON'T CARE

Field data in the Method Area

Per field:

- Name
- Type
- Modifiers
- Attributes

FIELD_INFO IN THE CLASS FILE

USED FOR THE RESOLUTION OF SYMBOLIC ADDRESSES SINCE THE TRANSLATION IS DYNAMIC
- STATIC RESOLUTION LANGUAGES CAN FORGET NAMES AND REASON ONLY WITH PHYSICAL ADDRESSES

PUBLIC, PRIVATE, ..

FOR ANNOTATIONS

FieldType descriptors

INSIDE CLASSFILE

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

THEN: [[MATRIX, [[[CUBE, ...

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

ALL THIS INFOS ARE NEEDED FOR METHODS EXECUTION!

THE METHOD CODE IN RUNTIME CONSTANT POOL CONTAINS FOR EACH METHOD

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)

COMPILER ESTIMATION OF SIZE OF THE OPERAND STACK AND LOCAL VARIABLES ARRAY

LATER ON

Per exception handler (one for each try/catch/finally clause)

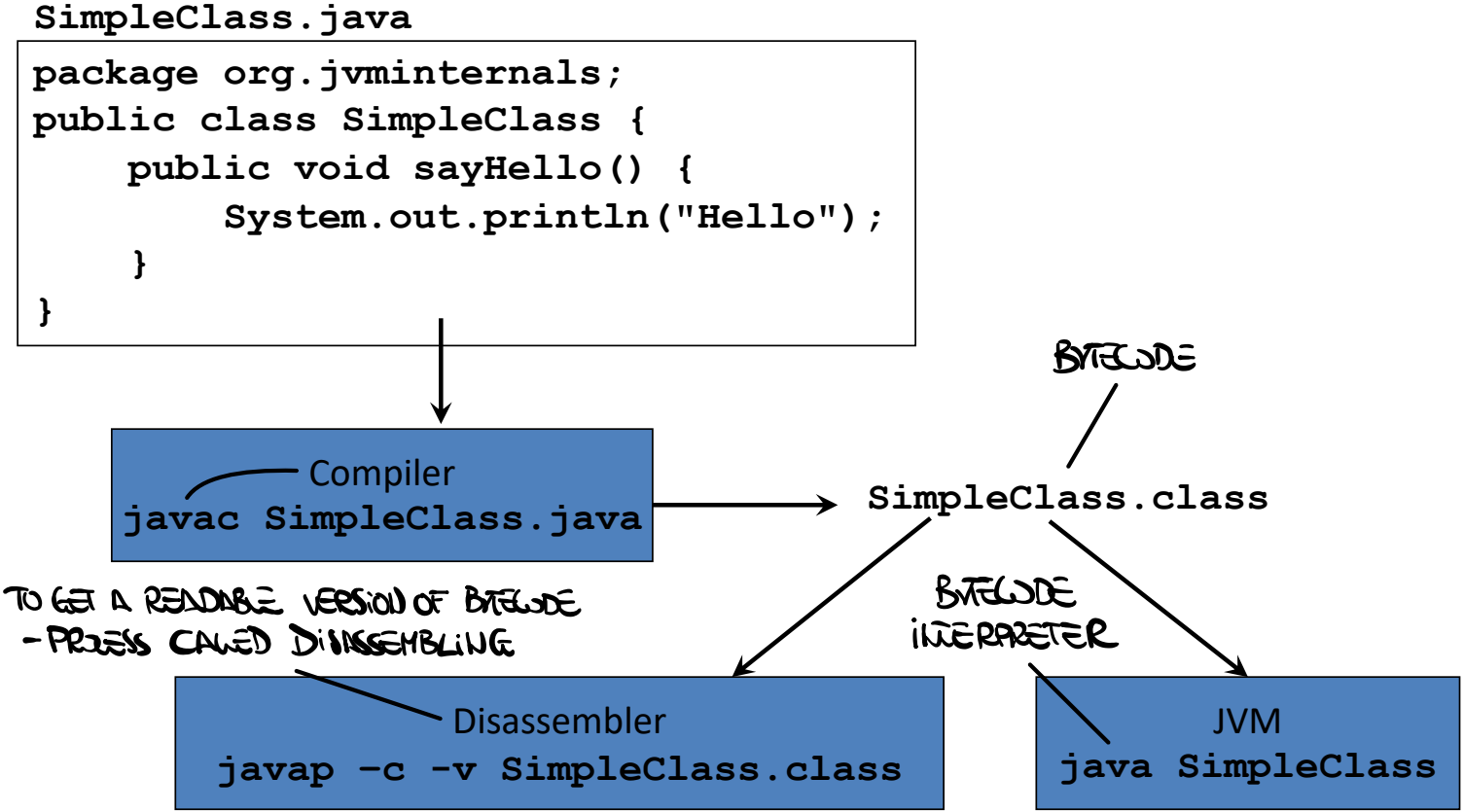
- Start point
- End point
- PC offset for handler code
- Constant pool index for exception class being caught

PC THEY + PC OFFSET = PC CATCH

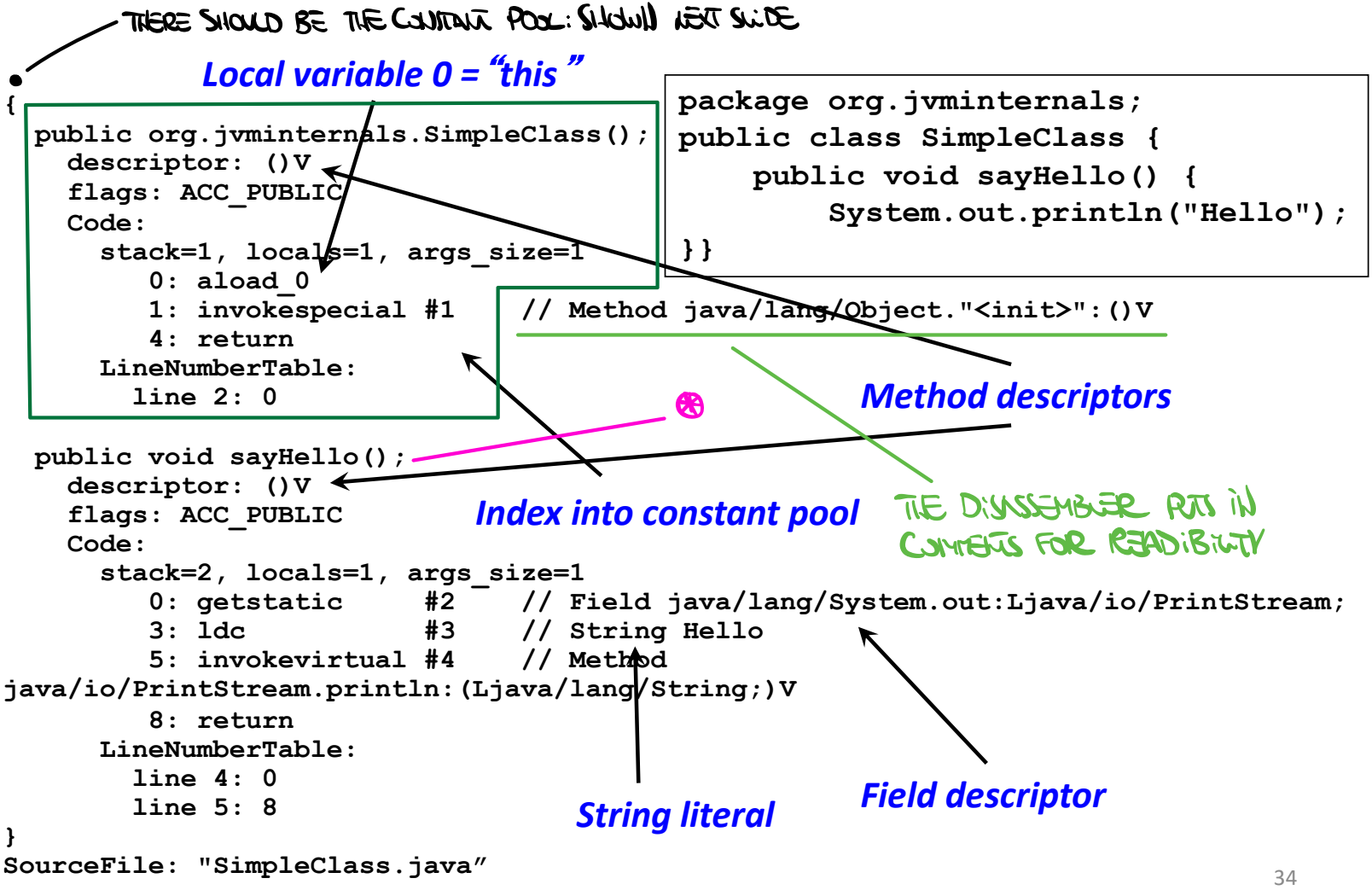
- THE EXCEPTION TABLE OF A METHOD SAYS
  - PROGRAM COUNTER OF THE START AND THE END OF THE BLOCK "TRY" IN THE BYTECODE
  - REFERENCE TO THE EXCEPTION CLASSES THAT THE CATCH BLOCK(S) CAPTURES
  - PC OFFSET IN THE BYTECODE TO JUMP TO THE HANDLER

- THE EXCEPTION HANDLING PROCEDURE IS AS FOLLOWS
  - IF AN EXCEPTION IS GENERATED BY AN ISTRUCTION (MAYBE A METHOD INVOCATION OR WHTVR) FIND THE CORRESPONDING TRY INTERVAL, FIND THE CORRESPONDING EXCEPTION TYPE AND THEN JUMP TO THE CATCH BYTECODE USING THE PC OFFSET SAW EARLIER

Disassembling Java files: javac, javap, java



SimpleClass.class: constructor and method



INFO ABOUT CLASS DESCRIPTOR

- METHOD DESCRIPTOR: ()V : NO PARAMS (), RETURNS VOID V
- FLAGS: ACC_PUBLIC, THE METHOD IS PUBLIC
- CODE
  - STACK: ESTIMATE SIZE OF THE OPERAND STACK (WHERE BYTECODE IS EXECUTED BY THE JVM)
  - LOCAL: LOCAL VARIABLE ARRAY SIZE GUESS (1, POS 0 = THIS)
  - ARGS_SIZE: # ARGS PASSED TO THE METHOD (1, JUST THIS)
  - UNDER THE INFO THERE IS THE ACTUAL BYTECODE
    - ALOAD_0: PUSH ON THE STACK THE ELEMENT IN POS 0 OF THE LOCAL VARIABLE ARRAY
      - THE ANALOGOUS OPERATION TO SAVE FROM STACK TO LOCAL VAR ARRAY IS STORE
    - INVOKESPECIAL #1 : INVOKE INSTANCE METHOD; SPECIAL HANDLING FOR SUPERCLASS, PRIVATE, ... #1 IS A SYMBOLIC REFERENCE TO THE CONSTANT POOL, WHEN RESOLVED IT WILL BE THE ADDRESS OF THE CONSTRUCTOR OF THE (SUPER)CLASS OBJECT
- LINENUMBERTABLE: DON'T CARE

SAME REASONING

IN COMPILERS THE SYMBOL TABLE MATCH TO EACH VARIABLE NAME SOME INFO (MEM. LOCATION, TYPE, ...)

The constant pool

THESE ARE THE TYPES THAT WE CAN FIND IN THE CONSTANT POOL OF THE .CLASS

- 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

OPS IN THE STACK USES LOGIC #INDEXES THAT, FOLLOWING A PATH ARE RESOLVED INTO PHYSICAL ADDRESS (SEE NEXT SLIDE)

STATIC CONSTANT POOL RESULTED BY THE COMPILATION OF SIMPLECLASS.JAVA

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
```

- HERE METHODS INFO (SAW BEFORE)

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

SOURCE CODE

PAIR OF ADDRESSES: CLASS AND NAMEANDTYPE OF THE FIELD

ALL THESE ADDRESSES ARE DEFINED BY THE COMPILER SO THAT CAN BE RESOLVED

STRING TO PRINT DIRECTLY IN THE .CLASS

BYTECODE

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

AT THIS POINT WE RESOLVE THE ADDRESS OF THE CLASS SYSTEM AND WE LOAD IN THE METHOD AREA THE CORRESPONDING INTERNAL REPRESENTATION (OF SYSTEM)

- GETSTATIC TO PUSH IN THE STACK THE STATIC FIELD (OUT) OF SYSTEM (AND ON (OUT WE WILL INVOKE PRINTLN)
  - IDC PUSH ON THE OPERAND STACK A REFERENCE CONTAINED IN THE CONSTANT POOL. IN THIS CASE THE REFERENCE POINTS TO THE STRING "HELLO" (FOLLOW THE PATH: #3 → #17 → "HELLO")

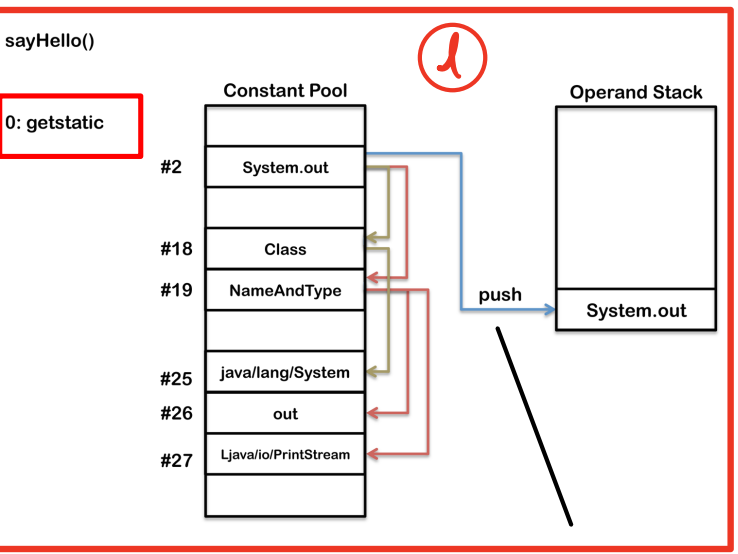
- (SEE BELOW)

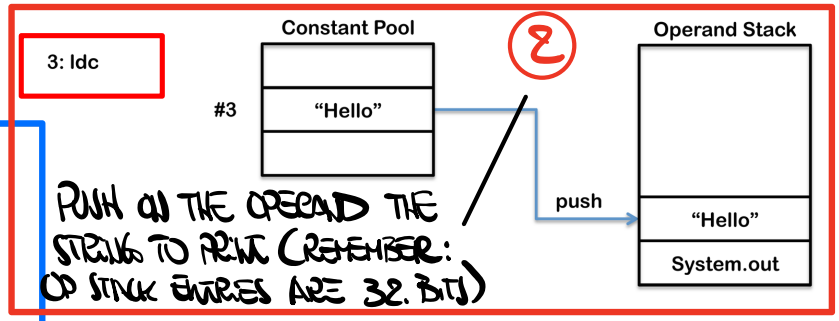
TO THE INVOKEVIRTUAL IT IS PASSED THE DESCRIPTOR OF THE FUNCTION PRINTLN, THE EFFECT OF INVOKEVIRTUAL IS TO CALL PRINTLN ON THE 2 (FIRST) VALUES ON THE STACK (WE PUSHED THIS=OUT AND THE PARAMETER "HELLO")

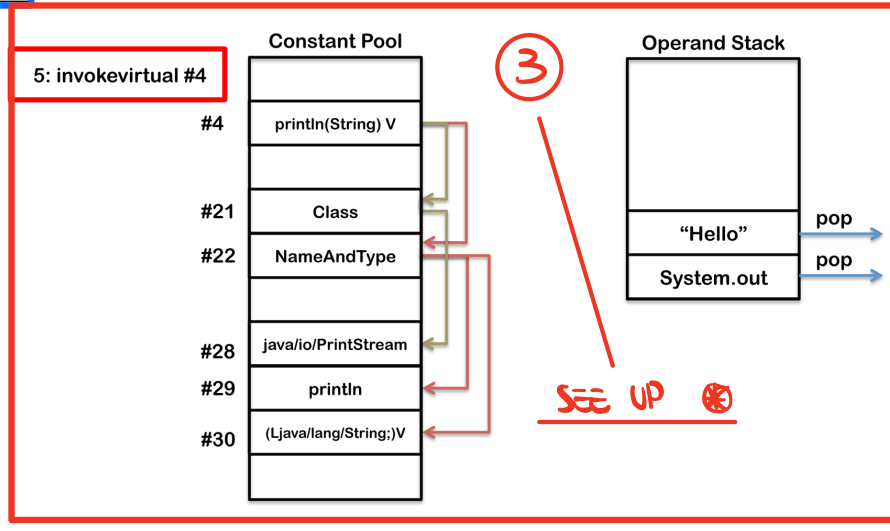
- THE EXECUTION PROCEEDS BY CREATING A NEW ACTIVATION RECORD IN THE THREAD AREA OF THE CURRENT THREAD, WHERE IN ITS OPERAND STACK WE HAVE OUT AND "HELLO". WHEN THE METHOD PRINTLN ENDS (ITS AR DESTROYED) WE WILL FIND THE RETURN VALUE ON THE STACK (IN AR) OF THE CALLER (SAYHELLO) (PRINTLN IS VOID SO IN THIS CASE NOTHING)

BYTECODE EXECUTION SIMULATION

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







GETSTATIC PUSH ON THE OPERAND STACK THE REFERENCE OF OUT (STATIC FIELD OF SYSTEM)

Loading, Linking, and Initializing

TRANSFORM .CLASS STATIC INFO (BINARY REP) IN INTERNAL REP INFO

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

THE .CLASS MIGHT ARRIVE FROM WEB

- **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>`

THERE COULD BE STATIC BLOCKS TO BE EXECUTED AT LOADING TIME (MAYBE TO INITIALIZE FIELDS OR WHTAR). THIS BLOCKS ARE CALLED CLINIT

CLASSIC PROCEDING OF THE PROGRAM STARTINGS

JVM Startup

MIND THAT THE POLICY OF CLASS LOADING DEPENDS ON THE SPECIFIC IMPLEMENTATION OF THE JVM

- The JVM starts up by loading an initial class using the **bootstrap classloader**
- The class is linked and initialized

THE BOOTSTRAP CLASSLOADER LOADS THE CLASS IN WHICH THE MAIN IS CONTAINED, IT GETS LINKED AND THEN

- **`public static void main(String[])`** is invoked.
- This will trigger loading, linking and initialization of additional classes and interfaces…

Loading

① PARSE THE BINARY DATA FROM .CLASS FILE ACCORDING TO SPECIFICATION, THEN TRADUCE IT IN THE IMPLEMENTATION DEPENDENT INTERNAL REP

② CREATES AN INSTANCE OF JAVA.LANG.CLASS

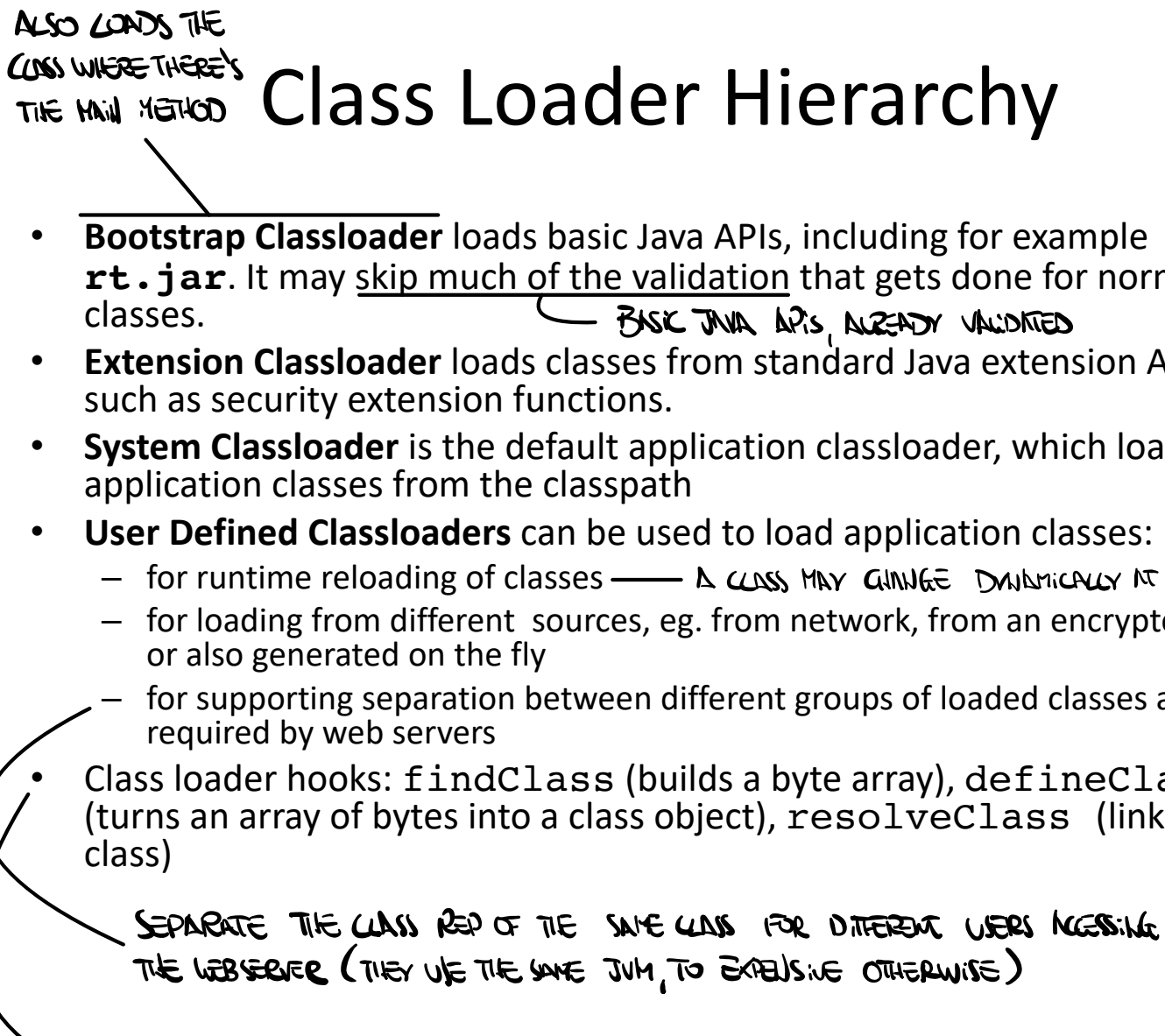
- Class or Interface *C* creation is triggered
  - by other class or interface referencing *C*
  - by certain methods (eg. reflection)
- Array classes are generated by the JVM
- Check whether already loaded
- If not, invoke the appropriate loader.loadClass
- Each class is tagged with the *initiating loader*

ARRAY CLASSES NOT LOADED BUT GENERATED ON USAGE BY THE JVM
- GENERATE AND LOAD AT USAGE FOR EACH ARRAY
- NEW CLASS IN CHANGE TYPE OR SIZE

THE REFERENCE IS IN THE CONS. POOL

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, eg. 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)



FIND CLASS: READS THE BINARY OF THE CLASS (FROM THE FILE OR ONLINE)
DEFINE CLASS: INTERPRETER THE BINARY TRANSFORMING IT IN A CLASS (IN TO INTERN REP)
RESOLVE CLASS: RESOLVE DYNAMICS REFERENCES

Runtime Constant Pool

THE RUNTIME CONST POOL IS THE INTERNAL REP OF THE STATIC CONST 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. (THEN ARE RESOLVED UPON USAGE)
- Symbolic references are derived from the `.class` file in the expected way
- Class names are those returned by **`Class.getName()`** AS PAR THE OBJ OF WHICH WE WANT THE CLASS
- Field and method references are made of name, descriptor and class name

A LINKER IN A PROGRAM THAT TAKES INDIVIDUAL COMPILED FILES AND COMBINES THEM INTO A SINGLE EXECUTABLE PROGRAM

Linking

OPTIONAL: THE RESOLUTION CAN BE EAGER OR LAZY
- IF DONE IN LINKING PHASE IS EAGER

- 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

ONCE THE VERIFICATION IS DONE THE JVM ALLOCATES THE MEMORY FOR THE CLASS VARIABLES AND INITIALIZE THEM TO THE DEFAULT VALUE ACCORDING TO THE TYPE OF VARIABLE.
- THE ACTUAL INITIALIZATION (WITH USER DEFINED VALUES) DO NOT OCCOUR UNTIL INITIALIZATION PHASE

Verification

ALSO CAN'T KNOW IF ALREADY VERIFICATED

AT RUNTIME

- 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 HAS SAFE POLICY: IF PASSED NO ERRORS, IF NOT PASSED THERE IS A CHANCE THAT STILL NOT ERROR

Verification Process

ONLY A SYNTACTIC STEP: CHECK THAT THE .CLASS IS OK BY THE CLASS FILE SPECIFICATION

- 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

SEMANTIC VERIFICATION STEP

  - 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.

OTHER CHECKS
- NO OVERFLOW OR UNDERFLOW
- VARIABLES ARE INITIALIZED
- TYPECHECKING ON METHOD PARS BASED ON SIGNATURES

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.

RUNTIME CHECKS
- BOTH CHECKS REQUIRES THE RESOLUTION OF DYNAMIC REFERENCE
  - DONE ON THE 1ST USAGE OF THE METHOD

(SYNTAX SPECIFIED BY THE JVM): STATIC METHOD THAT INITIALIZE TO STD VALUES THE STATIC FIELDS (IN ME Ø BY DEFAULT BEFORE THE USER INIT, FOR EXAMPLE)

Initialization

- <clinit> initialization method is invoked on classes and interfaces to initialize class variables
- static initializers are executed — AFTER INIT, DEFINED BY USER
- 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

KINDA CLINIT BUT FOR INSTANCES

INITIALIZERS MODIFY STATIC FIELDS (SHARED IN THE CLASS), SO IN THE CASE OF SHARED SUPERCLASSES (EG. OBJECT) WE HAVE TO SYNCHRONIZE (MUTUAL EXCLUSION, LOCKS) THE ACCESSES OF DIFFERENT INITIALIZERS

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 **`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

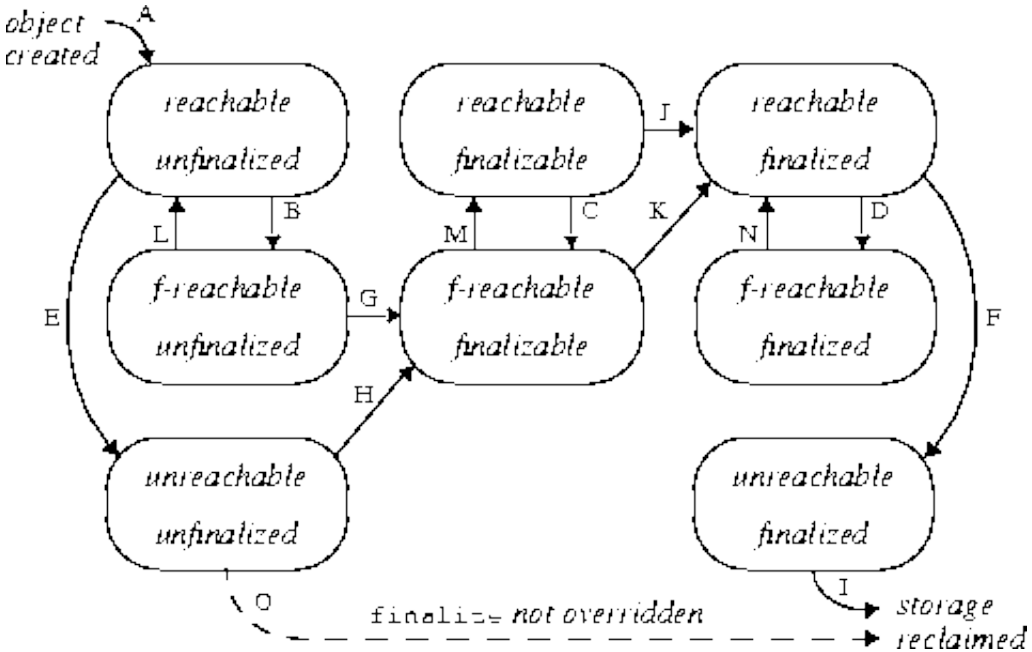
- Invoked just before garbage collection
- JLS does not specify when it is invoked
- Also does not specify which thread
- No automatic invocation of super's finalizers
- Very tricky!

```
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