JAVA VIRTUAL MACHINE

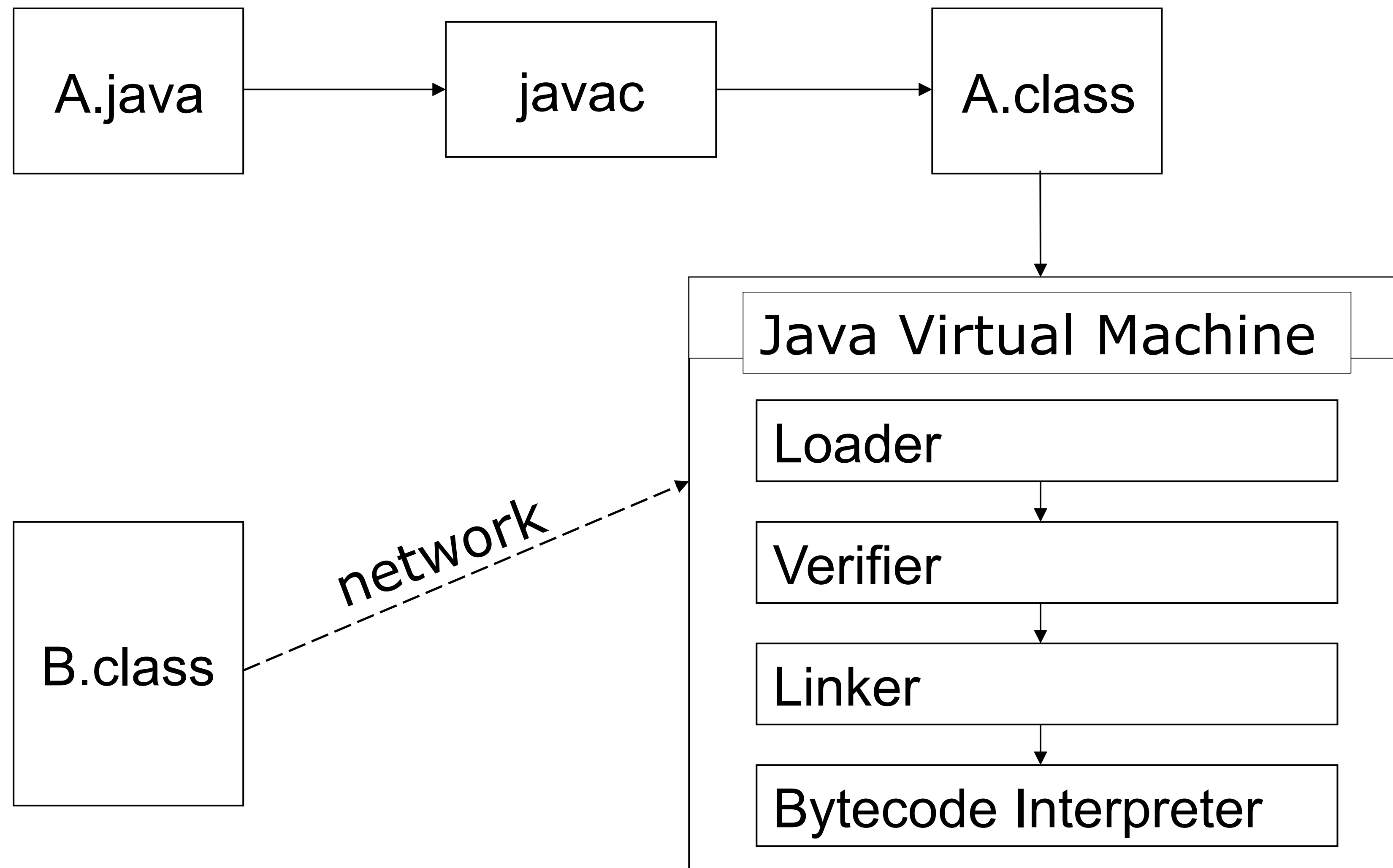
Java Run-Time

- Java Run-Time include tutte le strutture necessarie per eseguire una applicazione Java: bytedoe compilato
  - **JVM – Java Virtual Machine**
  - JCL – Java Class Library (Java API)
- The Java Virtual Machine Specification, Java SE 8
  - https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
  - Una review: http://blog.jamesdbloom.com/JVMInternals.html
- Java 9 (appena uscito sttemebre 2017)

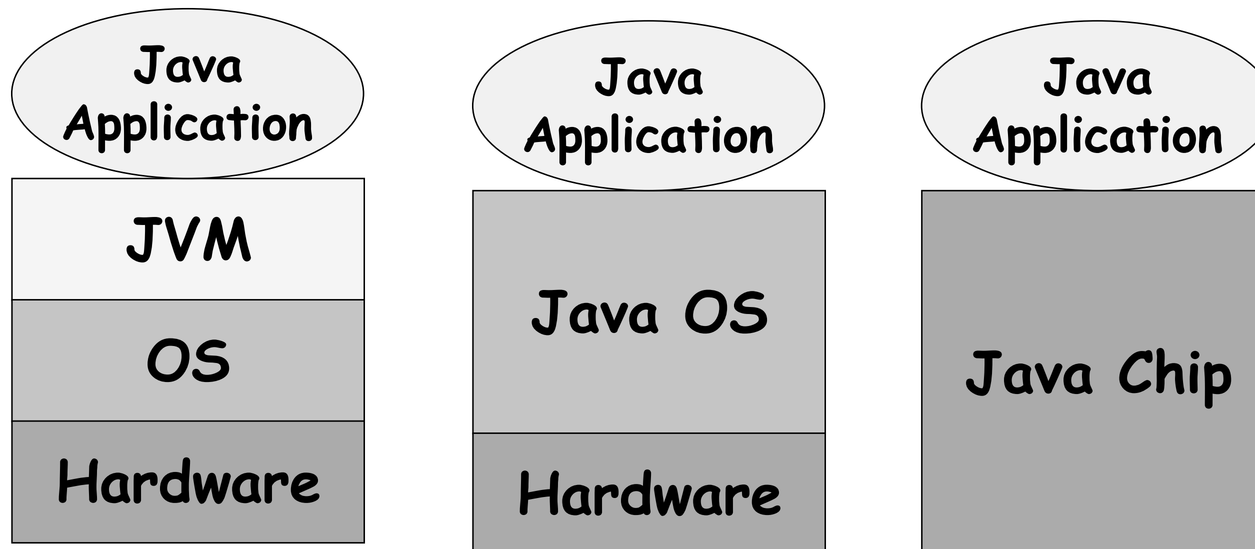
Cosa è la JVM

- LA JVM è la **macchina astratta** di Java
- JVM descrive "class file format" ovvero la struttura "machine independent" che tutte le implementazione della JVM devono suppportare.
- La JVM definisce I vincoli sintattici e strutturali che codici ifile .class devono rispettare
- La JVM ha un proprio linguaggio macchina: byte code
  - https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

JVM: visione di insieme



JVM: Implementazioni

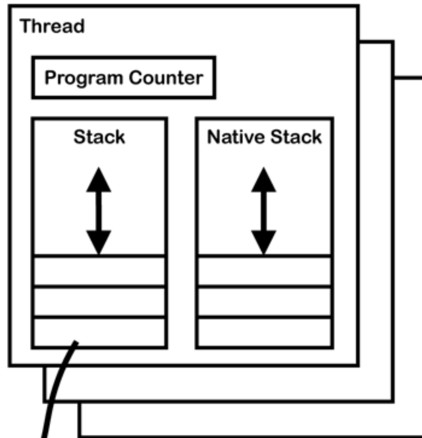


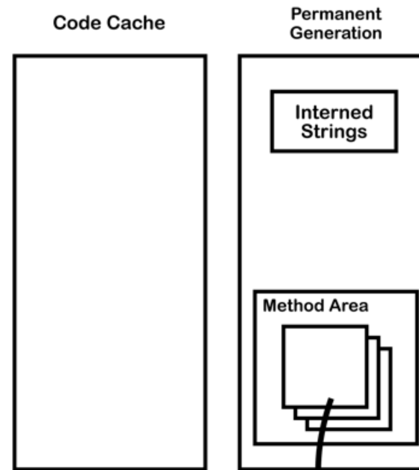
La specifica della JVM non prescrive come deve essere implementata

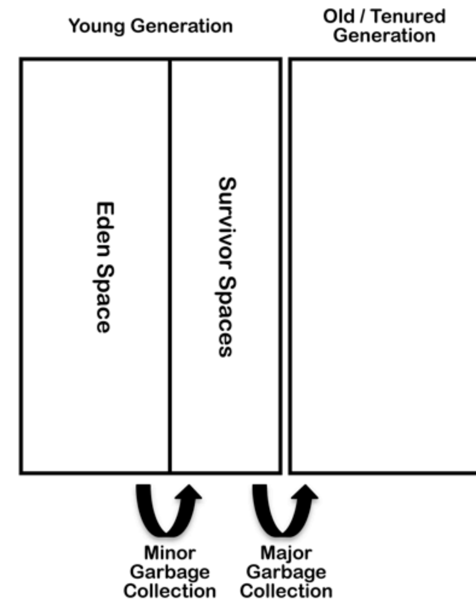
JVM

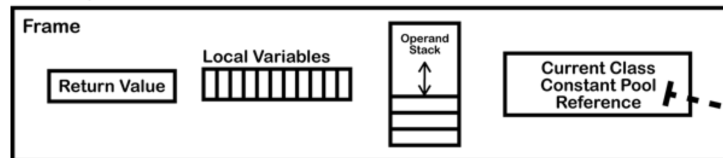
- JVM = macchina astratta "stack-based" multi-hreaded
- Istruzioni (bytecode) si basano uno stack:
  - Gli argomenti di una istruzione si trovano in testa allo stack
  - Il risultato dell'aoperazione viene messo in testa allo stack
- Lo stack degli operandi viene utilizzato
  - Trasmissione dei parametri ai metosi
  - Restituzione del risultato dell'invocazione di un metodo
  - Memorizzazione dei risultati intermedi
  - Memorizzazione delle variabili locali.

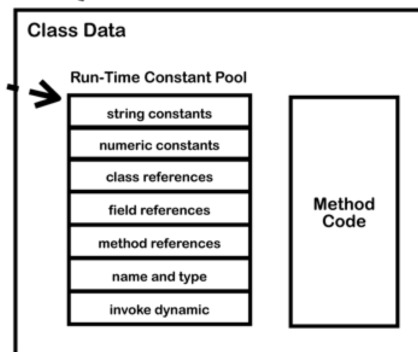
Stack **Non Heap** **Heap**

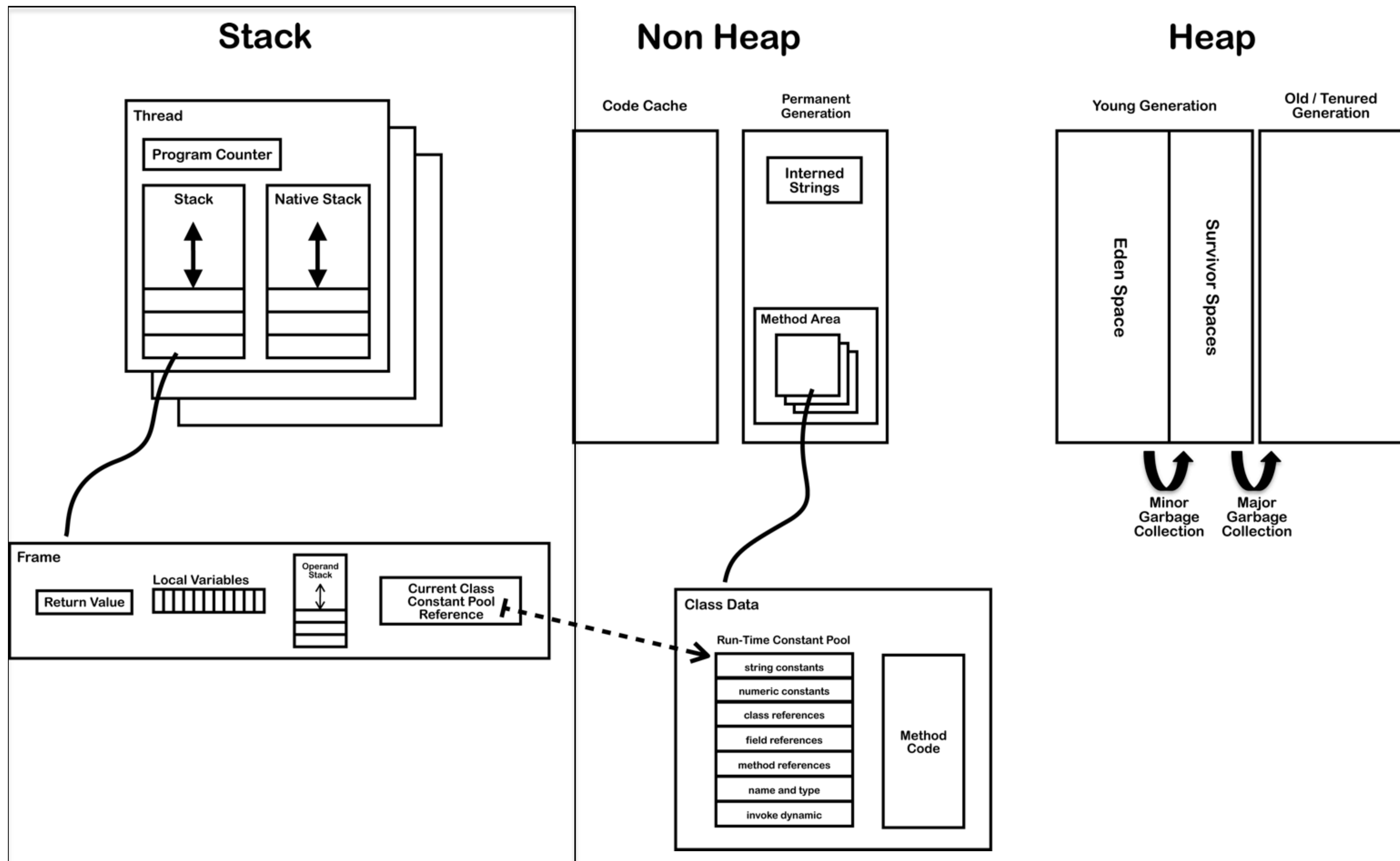












Thread

Condiviso tra i thread

Thread

- **Program Counter (PC):** Indirizzo dell'istruzione corrente del metodo in esecuzione (controllo sequenza)
  - PC sostanzialmente un indirizzo di memora della Method Area.
- Stack che contiene i record di attivazione (frame) dei metodi (ambiente). Struttura del Java Frame:
  - Local variable array
  - Return value
  - Operand stack
  - Reference to runtime info (descrittori di tipo)

Variabili Locali

- Local Variable Array: contiene tutte le variabili locali del metodo e tutti i parametri incluso **this**.
  - Metodi statici: le variabili locali partono dall'indice 0.
  - Metodi di istanza (non statici) la posizione 0 è utilizzata per la gestione di **this**.

Variabili locali (tipi)

- boolean
- byte
- char
- **long**
- short
- int
- float
- **double**
- reference
- returnAddress

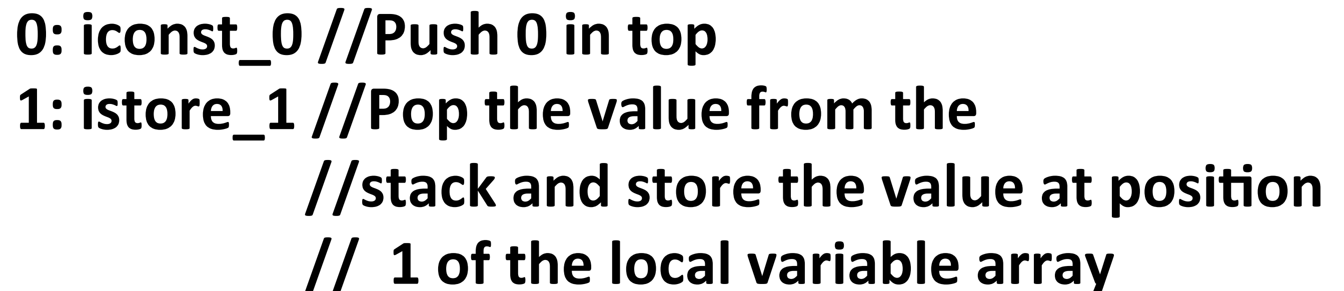
- Long & Double 64 bit
- Tutto il resto 32 bit

Operand Stack

Lo stack degli operandi è utilizzato dalle istruzioni del bytecode per la gestione degli argomenti e risultati delle operazioni

Compilazione

int i = 0



Intermezzo

Stack Abstract Machine

Stack Machine

Instruction	Stack Before		Stack Later
$STCstInt(n)$	s	$\longrightarrow$	s, n
$STAdd$	s, n_1, n_2	$\longrightarrow$	$s, (n_1 + n_2)$
$STSub$	s, n_1, n_2	$\longrightarrow$	$s, (n_1 - n_2)$
$STMul$	sn_1, n_2	$\longrightarrow$	$s, (n_1 * n_2)$
$STDup$	s, n	$\longrightarrow$	s, n, n
$STSwap$	s, n_1, n_2	$\longrightarrow$	s, n_2, n_1

Istruzioni operano sullo stack

Stack = controllo trasferimento dati

Interprete per Stack Machine

```
type stkinstr =
   | STCstInt of int
   | STAdd
   | STSub
   | STMul
   | STDup
   | STSwap
```

Sintassi astratta

Codice Interprete

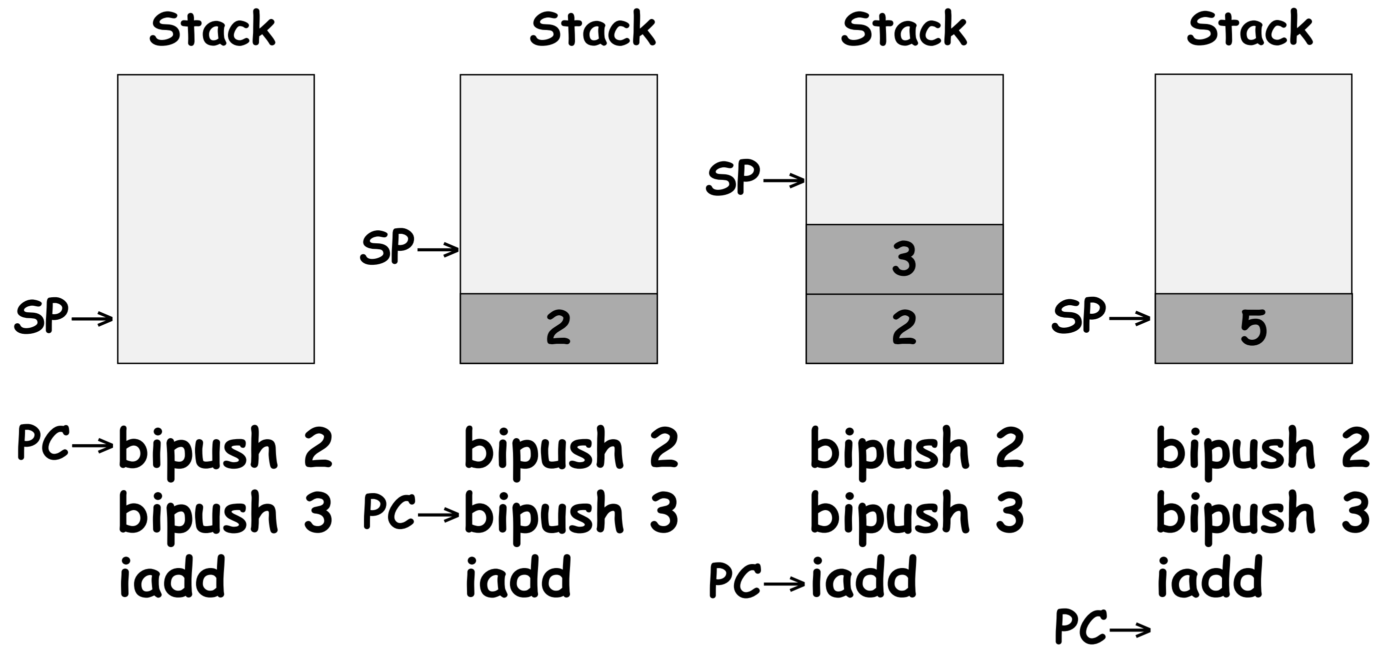
```
let rec stkEval (control: stkinstr list)
                 (stack: int list): int =
 match (control, stack) with
    | ([], v::_) -> v
    |( [], []) -> failwith("no result on the stack")
    | (STCstInt(n)::cs, stack)-> stkEval cs (n::stack)
    | (STAdd::cs, n1::n2::ss) -> stkEval cs ((n1+n2)::ss)
    | (STSub::cs, n1::n2::ss) -> stkEval cs ((n1-n2)::ss)
    | (STMul::cs, n1::n2::ss) -> stkEval cs ((n1*n2)::ss)
    | (STDup::cs, n::ss) -> stkEval cs (n::n::ss)
    | (STSwap::cs, n1::n2::ss) -> stkEval cs (n2::n1::ss)
    | (_::_, []) -> failwith("too few operands")
```

REPL

```
(* The REPL Usage *)

# let c =
STCstInt(5)::STCstInt(6)::STAdd::STCstInt(8)::STSub::[];;
val c : stkinstr list = [STCstInt 5; STCstInt 6; STAdd;
STCstInt 8; STSub]
# stkEval c [];;
- : int = -3
```

Esempio: bytecode



Linking

- Durante la fase di **linking** i riferimenti simbolici presente nel codice oggetto vengono sostituiti con un indirizzo di memoria **reale** relativo all'eseguibile finale.

- Nel caso di C / C ++ la fase di linking avviene tramite i DLL.

Linking Dinamico

- Ogni frame contiene un puntatore a una struttura denominata **Constant Pool**: descrittore di tipo associato alla classe dove è definito il metodo in esecuzione.

- Meccanismo utilizzato in Java per il linking dinamico

Linking Dinamico

- Quando viene compilata una classe Java, tutti i riferimenti a variabili e metodi presenti nel codice della classe vengono memorizzati nella constant pool associata alla classe come riferimenti simbolici.
- Un riferimento simbolico è un riferimento logico e non un riferimento che indica effettivamente una posizione di memoria fisica.

Risoluzione dei riferimenti simbolici

- Risoluzione a loading time: **eager resolution**
- Risoluzione efftuata la prima volta che durante l'esecuzione si incontra un link simbolico: **lazy resolution (JVM)**
- Se il riferimento simbolico si riferisce a una classe che non è stata ancora caricata, questa classe verrà caricata (**Lazy class loading**)
- Ogni riferimento risolto diviene un offset rispetto alla struttura di memorizzazione a runtime.

Method Area

- Porzione della memoria dove sono memorizzate le classi. Per ogni classe:
- Riferimento al Classloader
- Per ogni classe:
  - Run Time Constant Pool
  - Field data
  - Method data
  - Method code

I file .class

- Il bytecode generato dal compilatore Java viene memorizzato in un ***class file*** (.class) contenente
  - **bytecode** dei metodi della classe
  - **constant pool**: una sorta di tabella dei simboli che descrive le costanti e altre informazioni presenti nel codice della classe

- Per vedere il bytecode basta usare

```
javap <class_file>
```

ClassFile {

u4 magic;	**0xCAFEBABE**
u2 minor_version; u2 major_version;	**Java Language Version**
u2 constant_pool_count; cp_info contant_pool[constant_pool_count–1];	**Constant Pool**
u2 access_flags;	**access modifiers and other info**
u2 this_class; u2 super_class;	**References to Class and Superclass**
u2 interfaces_count; u2 interfaces[interfaces_count];	**References to Direct Interfaces**
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**

}

SimpleClass.java

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

Compiler

javac SimpleClass.java

SimpleClass.class

Disassembler

javap -c -v SimpleClass.class

JVM

java SimpleClass

```java
package org.jvminternals;

public class SimpleClass {

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

}
```

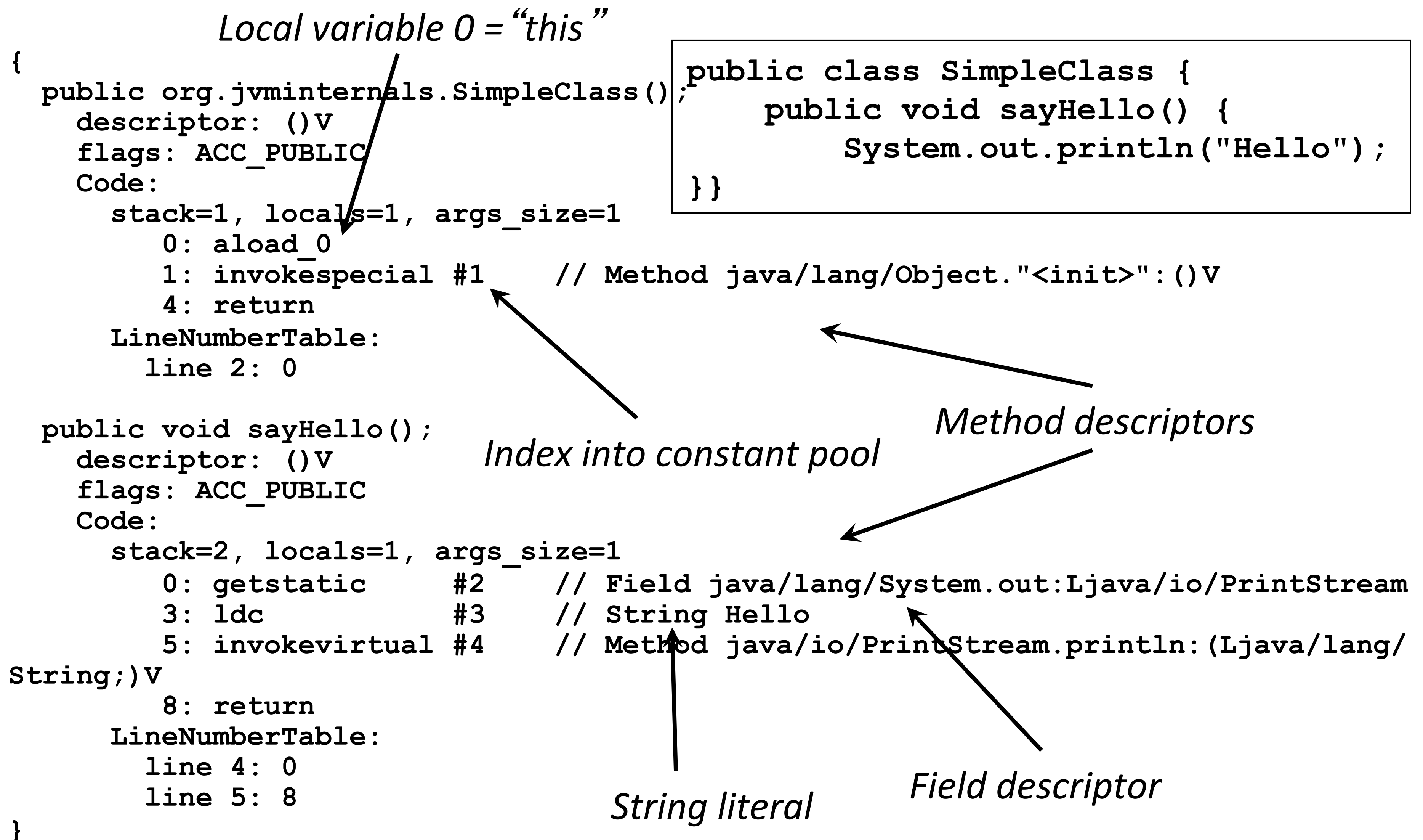
```
javap -v -p -s -sysinfo –constants
   org/jvminternals/SimpleClass.class
```

```
public class org.jvminternals.SimpleClass
  SourceFile: "SimpleClass.java"
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#17         //  java/lang/Object."<init>":()V
   #2 = Fieldref           #18.#19        //  java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #20            //  "Hello"
   #4 = Methodref          #21.#22        //  java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #23            //  org/jvminternals/SimpleClass
   #6 = Class              #24            //  java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               Lorg/jvminternals/SimpleClass;
:
:
```

```
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
```

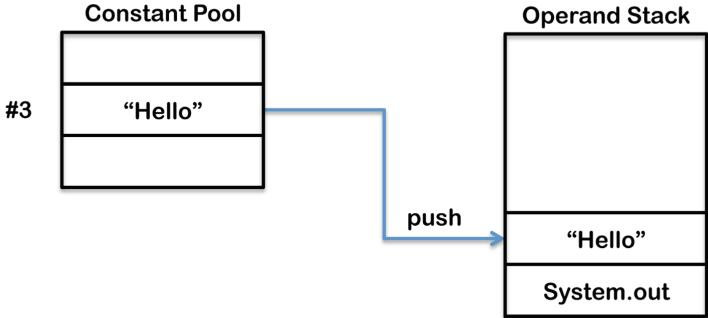
```
public class SimpleClass {
    public void sayHello() {
        System.out.println("Hello");
}}
```

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



Local variable 0 = "this"
{
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
}
public class SimpleClass {
public void sayHello() {
System.out.println("Hello");
}}
Index into constant pool
Method descriptors
String literal
Field descriptor

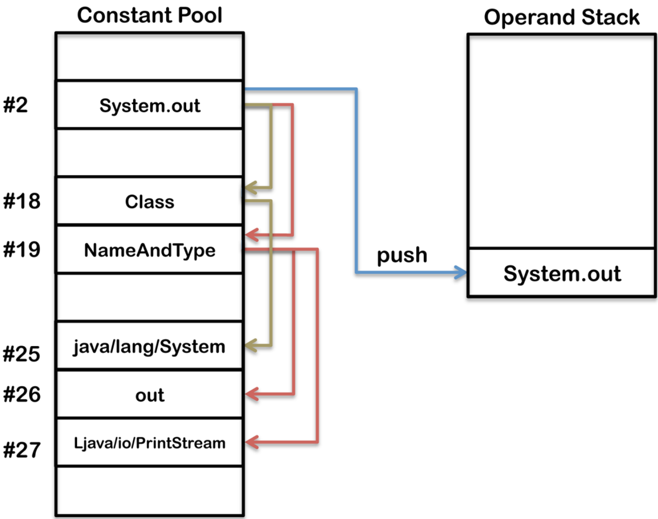
3: ldc



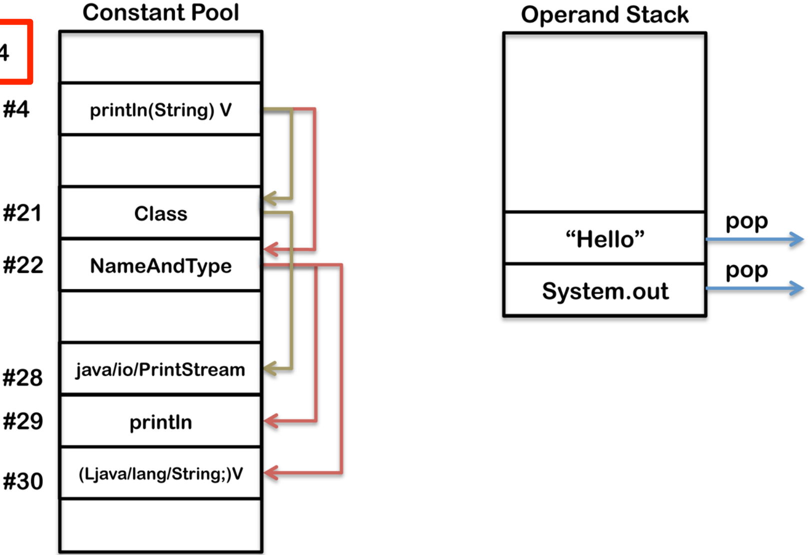

```
public void sayHello();
   descriptor: ()V
   Code:
     stack=2, locals=1, args_size=1
        0: getstatic      #2
        3: ldc            #3
        5: invokevirtual #4
        8: return
```

sayHello()

0: getstatic



5: invokevirtual #4



.class: esempio

```
public class Foo {

   public static void main (String args[]) {
     System.out.println("Programmazione 2");
   }
 }
```

```
javac Foo.java // Foo.class

javap –c –v Foo
```

Constant pool (I)

```
Constant pool:
const #1 = Method            #6.#15;  //  java/lang/Object."<init>":()V
const #2 = Field   #16.#17;  //  java/lang/System.out:Ljava/io/PrintStream;
const #3 = String  #18;      //  Programmazione 2
const #4 = Method            #19.#20;//  java/io/PrintStream.println:(Ljava/lang/
String;)V
const #5 = class   #21;      //  Foo
const #6 = class   #22;      //  java/lang/Object
const #7 = Asciz   <init>;
const #8 = Asciz   ()V;
const #9 = Asciz   Code;
const #10 = Asciz LineNumberTable;
const #11 = Asciz main;
const #12 = Asciz ([Ljava/lang/String;)V;
const #13 = Asciz SourceFile;
const #14 = Asciz Foo.java;
const #15 = NameAndType            #7:#8; //  "<init>":()V
const #16 = class #23;       //  java/lang/System
```

Constant pool (I)

```
Constant pool:
const #1 = Method           #6.#15;  //  java/lang/Object."<init>":()V
const #2 = Field   #16.#17;  //  java/lang/System.out:Ljava/io/PrintStream;
const #3 = String  #18;      //  Programmazione 2
const #4 = Method           #19.#20;//  java/io/PrintStream.println:(Ljava/lang/
String;)V
const #5 = class   #21;      //  Foo
const #6 = class   #22;      //  java/lang/Object
const #7 = Asciz   <init>;
const #8 = Asciz   ()V;
const #9 = Asciz   Code;
const #10 = Asciz LineNumberTable;
const #11 = Asciz main;
const #12 = Asciz ([Ljava/lang/String;)V;
const #13 = Asciz SourceFile;
const #14 = Asciz Foo.java;
const #15 = NameAndType             #7:#8; //  "<init>":()V
const #16 = class #23;      //  java/lang/System
```

riferimenti simbolici

Constant Pool (II)

```
const #17 = NameAndType        #24:#25;//  out:Ljava/io/
PrintStream;
const #18 = Asciz Programmazione 2;
const #19 = class #26;     //  java/io/PrintStream
const #20 = NameAndType        #27:#28;//println:(Ljava/lang/
String;)V
const #21 = Asciz Foo;
const #22 = Asciz java/lang/Object;
const #23 = Asciz java/lang/System;
const #24 = Asciz out;
const #25 = Asciz Ljava/io/PrintStream;;
const #26 = Asciz java/io/PrintStream;
const #27 = Asciz println;
const #28 = Asciz (Ljava/lang/String;)V;
```

A cosa serve la constant pool?

- La constant pool viene utilizzata nel class loading durante il processo di risoluzione
  - quando durante l'esecuzione si fa riferimento a un nome per la prima volta questo viene risolto usando le informazioni nella constant pool
  - le informazioni della constant pool permettono, ad esempio, di caricare la classe dove il nome è stato definito

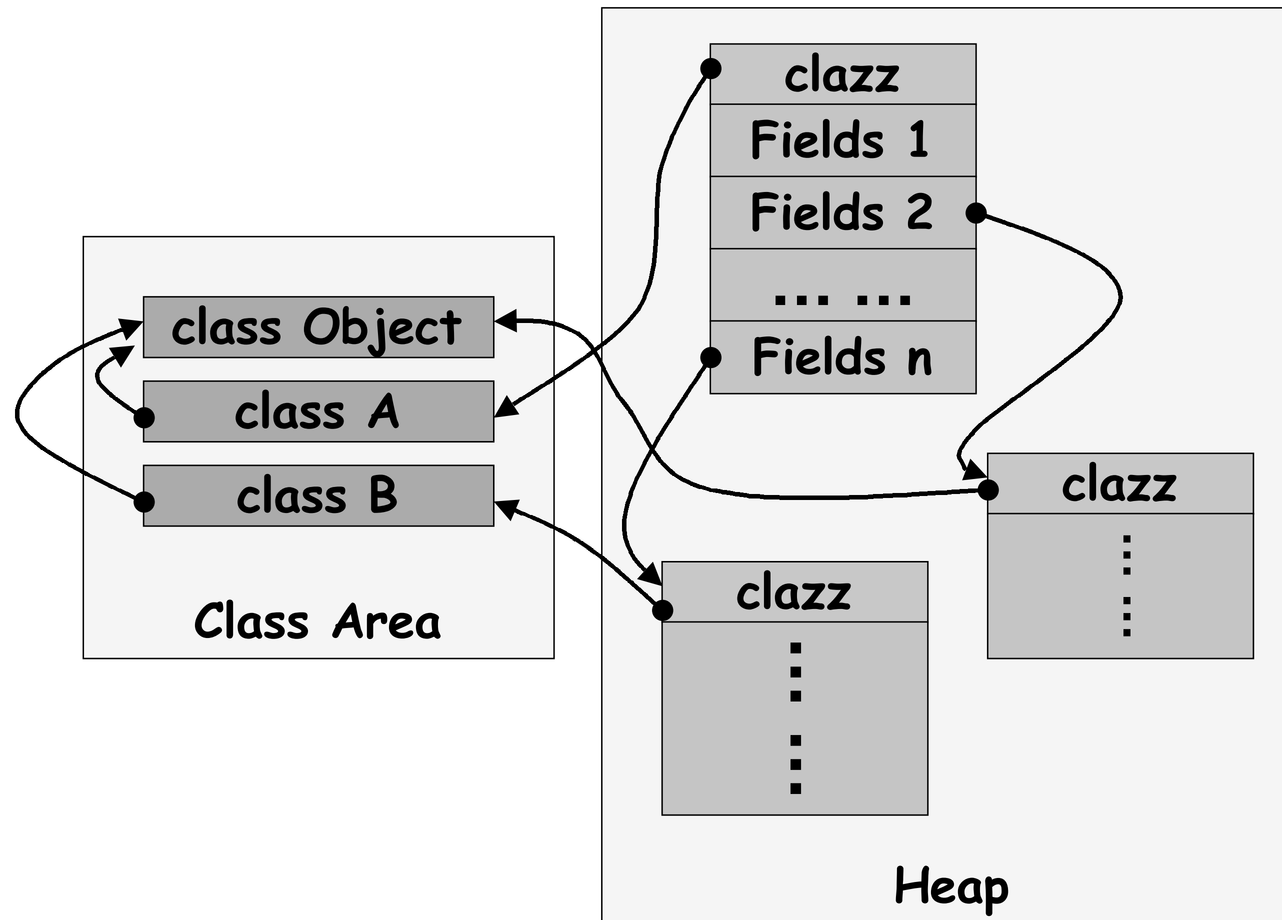
Esempio

```
public class Main extends java.lang.Object  SourceFile: "Main.java"
minor version: 0
major version: 50
Constant pool:
const #1 = Method #9.#18;//  …
const #2 = class #19;//  Counter
const #3 = Method #2. #18;//  Counter."<init>":()V
:
const #5 = Method#2.#22;//  Counter.inc:()I
const #6 = Method#23.#24;
const #7 = Method#2.#25;//  Counter.dec:()I
const #8 = class#26;//  Main
```

```
class Counter {
   int inc( ) { .. }
   int dec( ) { .. }
}
```

- **La name resolution permette di scoprire che inc e dec sono metodi definiti nella classe Counter**
  - **viene caricata la classe Counter**
  - **viene salvato un puntatore**

Oggetti e Heap



E i metodi?

- I metodi di classi Java sono rappresentati in strutture simili alle vtable di C ++
- Ma gli offset di accesso ai metodi della **vtable non sono determinati staticamente**
- Il valore dell'offset di accesso viene calcolato dinamicamente la prima volta che si trova un riferimento all'oggetto
- Un eventuale secondo accesso utilizza l'offset

Esaminiamo nel dettaglio la procedura di accesso ai metodi

JVM è una stack machine

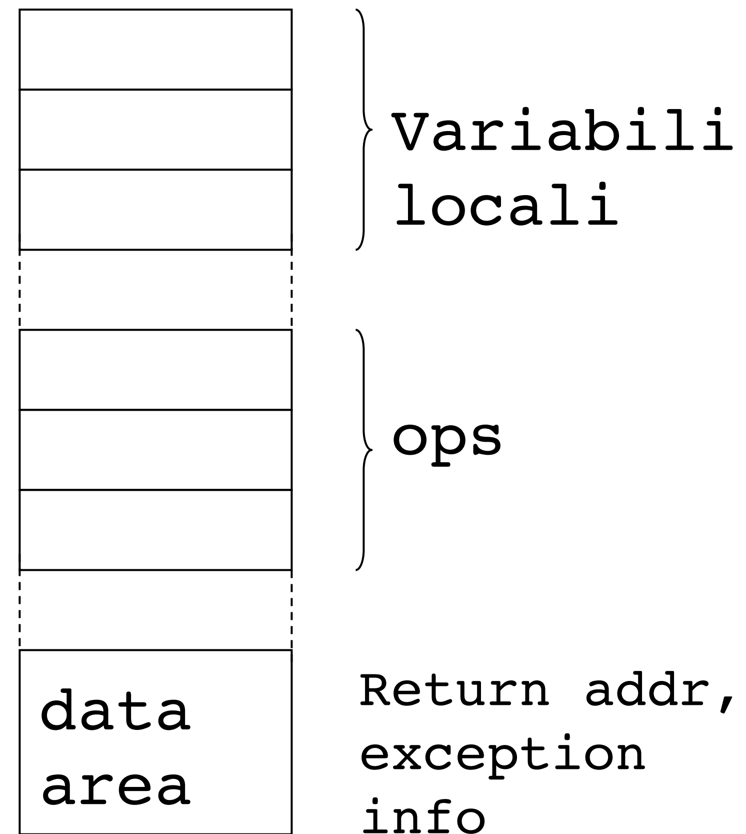
- Java

```
class A extends Object {
    int i;
    void f(int val) { i = val + 1;}
}
```

- Bytecode

```
Method void f(int)
    aload 0    ; object ref this
    iload 1    ; int val
    iconst 1
    iadd       ; add val +1
    putfield #2  // Field i:int
    return
```

riferimento alla const pool



Esempio

- Codice di un metodo

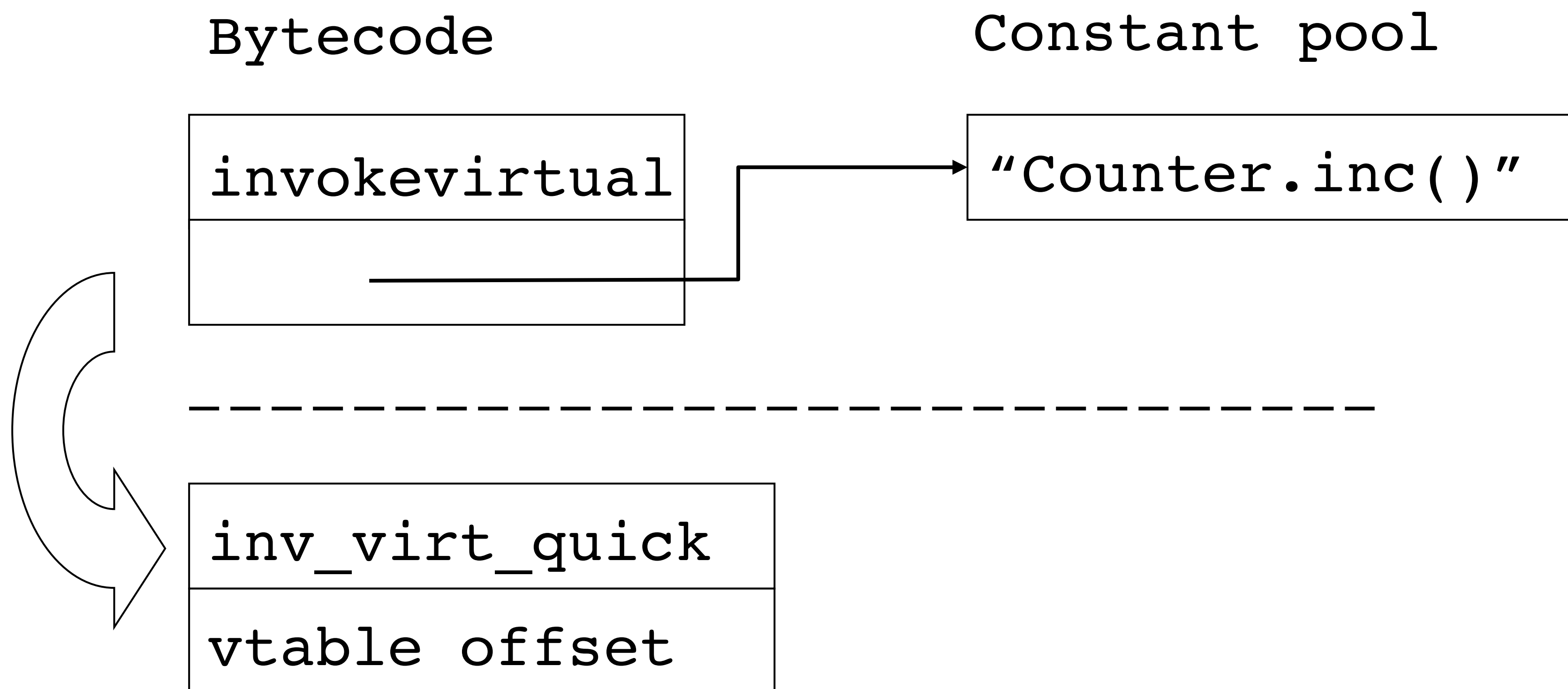
  ```
  void add2(Incrementable x) { x.inc(); x.inc(); }
  ```

- Ricerca del metodo
  - trovare la classe dove il metodo è definito
  - trovare la vtable della classe
  - trovare il metodo nella vtable
- Chiamata del metodo
  - creazione del record di attivazione, …

Bytecode

```
public static void main(java.lang.String[]);
Code:
 Stack=2, Locals=2, Args_size=1
 0:   new             #2; //class Counter
 3:   dup
 4:   invokespecial  #3;//Method Counter."<init>":()V
 7:   astore_1
 8:   getstatic      #4;//Field java/lang/System.out:Ljava/io/PrintStream;
 11:  aload_1
 12: invokevirtual      #5; //Method Counter.inc:()I
 15:  invokevirtual  #6; //Method java/io/PrintStream.println:(I)V
 18:  getstatic      #4; //Field java/lang/System.out:Ljava/io/PrintStream;
 21:  aload_1
 22:  invokevirtual  #7; //Method Counter.dec:()I
 25:  invokevirtual  #6; //Method java/io/PrintStream.println:(I)V
 28:  return
```

Bytecode : invokevirtual



- Dopo la ricerca si possono utilizzare offset calcolati la prima volta (senza overhead di ricerca)

Java interface

L'offset del metodo foo è diverso nelle due tabelle

```
interface I {
  void foo();
}

public class A implements I {
  :
  void foo() { .. }
  :
}

public class B implements I {
  :
  void m() { .. }
  void foo() { .. }
  :
}
```

Bytecode: invokeinterface

Bytecode | Constant pool

invokeinterface

→ "A.foo()"

---

inv_int_quick

→ "A.foo()"

Secondo accesso: tramite l'offset determinato si controlla la presenza del metodo altrimenti si effettua la ricerca come la prima volta

Leggiamo la documentazione della JVM

Method invocation:

- **invokevirtual:** usual instruction for calling a method on an object
- **invokeinterface:** same as invokevirtual, but used when the called method is declared in an interface (requires different kind of method lookup)
- **invokespecial:** for calling things such as constructors. These are not dynamically dispatched (this instruction is also known as invokenonvirtual)
- **invokestatic:** for calling methods that have the “static” modifier (these methods “belong” to a class, rather an object)

Returning from methods:

return, ireturn, lreturn, areturn, freturn, …

JVM: tabelle degli oggetti

```
public abstract class AbstractMap<K,V> implements Map<K,V> {
  Set<K> keySet;
  Collection<V> values;
}
public class HashMap<K,V> extends AbstractMap<K,V> {
  Entry[] table;
  int size;
  int threshold;
  float loadFactor;
  int modCount;
  boolean useAltHashing;
  int hashSeed
}
```

KeySet è il primo campo della tabella? Table il terzo?

La struttura effettiva

```
java -jar target/java-object-layout.jar java.util.HashMap
   java.util.HashMap
offset size type description
0 12 (object header + first field alignment)
12 4 Set AbstractMap.keySet
16 4 Collection AbstractMap.values
20 4 int HashMap.size
24 4 int HashMap.threshold
28 4 float HashMap.loadFactor
32 4 int HashMap.modCount
36 4 int HashMap.hashSeed
40 1 boolean HashMap.useAltHashing
41 3 (alignment/padding gap)
44 4 Entry[] HashMap.table
48 4 Set HashMap.entrySet
52 4 (loss due to the next object alignment)
56 (object boundary, size estimate
VM reports 56 bytes per instance
```

Ordine di strutturazione

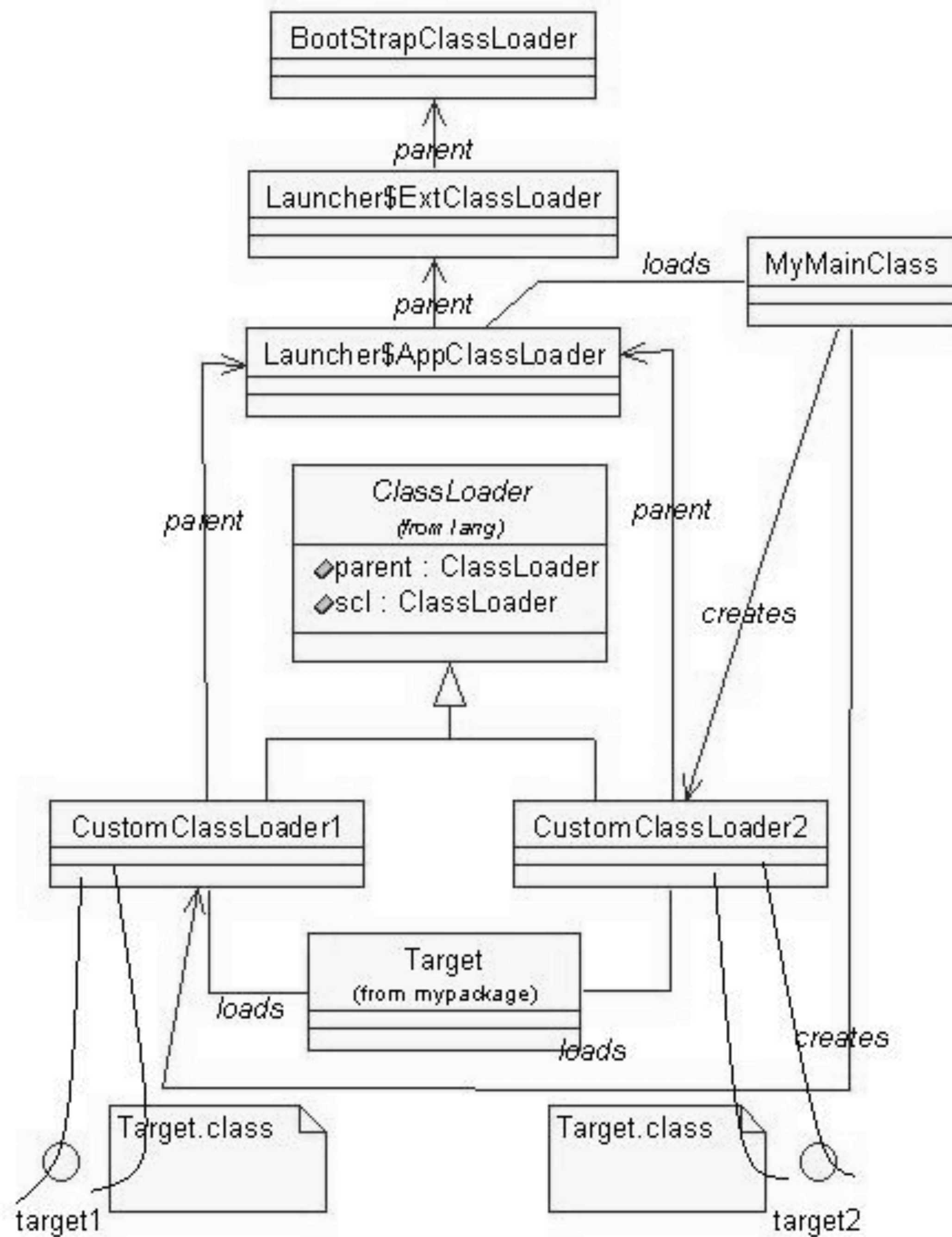
1) doubles e longs
2) ints e floats
3) shorts e chars
4) booleans e bytes
5) references

JVM Internals

- Scaricate e eseguite gli esempi definiti nel progetto **OPENJDK** (http://openjdk.java.net)
- In particolare: **jol** *(Java Object Layout) is the tiny toolbox to analyze object layout schemes in JVMs. These tools are using Unsafe heavily to deduce the* ***actual object layout and footprint. This makes the tools much more accurate than others relying on heap dumps, specification assumptions, etc.***

Class loading in Java

- Una classe è caricata e inizializzata quando un suo oggetto (o un oggetto che appartiene a una sua sottoclasse) è referenziato per la prima volta
- JVM loading = leggere il class file + verificare il bytecode, integrare il codice nel run-time



Visione complessiva

Inizializzazione

```
class A {
 static int a = B.b + 1; // codice a run-time
                         // A.<clinit>
}

class B {
 static int b = 42; // codice a run-time
                    // B.<clinit>
}
```

L'inizializzazione di A è sospesa: viene terminata quando B è inizializzato

Inizializzazione: Bytecode

```
class A {
  String name;
  A(String s) {
    name = s;
  }
}
```

```
<init>(java.lang.String)V
0: aload_0  //this
1: invokespecial java.lang.Object.<init>()V
4: aload_0  //this
5: aload_1  //parameter s
6: putfield A.name
9: return
```

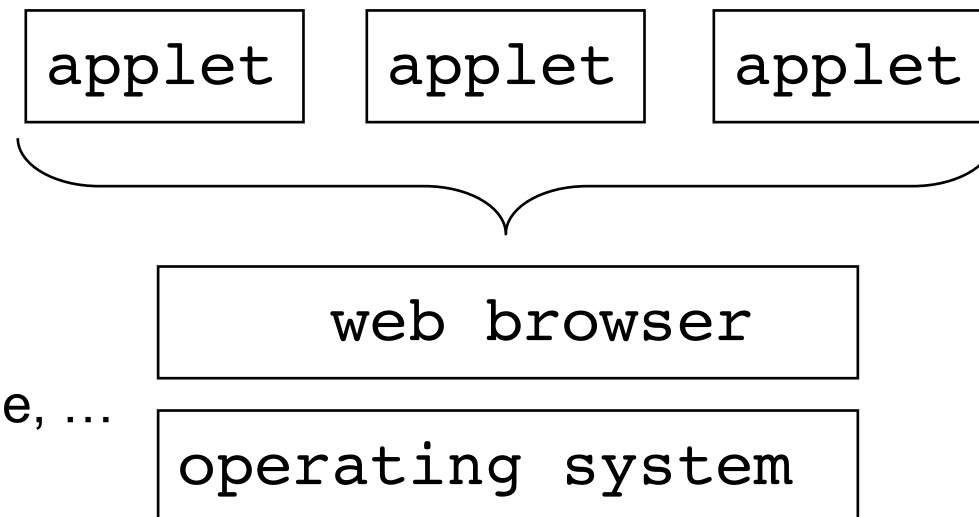
JVM interpreter

```
do {
   byte opcode = fetch an opcode;
   switch (opcode) {
     case opCode1 :
           fetch operands for opCode1;
           execute action for opCode1;
           break;
     case opCode2 :
           fetch operands for opCode2;
           execute action for opCode2;
           break;
     case ...
} while (more to do)
```

Java stack inspection

Codice Mobile

- Java: progettato per codice mobile



- SmartPhone, …

Applet security

- Protezione risorse utente
- Cosa non deve poter fare una applet
  - mandare in crisi il browser o il SO
  - eseguire **“rm –rf /”**
  - usare tutte le risorse del sistema
- Cosa deve poter fare una applet
  - usare alcune risorse (ad esempio per far vedere una figura sul display, oppure un gioco)…
  - … ma in modo isolato e protetto
- In sicurezza questo viene denominato: principio del minimo privilegio

Java (ma vale anche per C#)

- Sistemi di tipo statici
  - garantiscono memory safety (non si usa memoria non prevista)
- Controlli a run-time
  - array index
  - downcast
  - verifica degli accessi
- Virtual machine
  - bytecode verification
- Garbage collection
  - lo vediamo la prossima lezione
  - crittografia, autenticazione (lo vedrete in altri insegnamenti…)

lo vediamo oggi

Controllo degli accessi

- Fornitori di servizio hanno livelli di sicurezza differenti (classico dei SO)
  - www.l33t-hax0rs.com vs. www.java.sun.com (ci fidiamo?)
  - untrusted code vs trusted code
- Trusted code può invocare untrusted code
  - e.g. invocare una applet per visionare dei dati
- Untrusted code può invocare trusted code
  - e.g. la applet può caricare una font specifica
- Quali sono le politiche per il controllo degli accessi?

Java Security Model

VM Runtime

Security Policy

a.class
b.class
c.class
d.class
e.class

Classloader
SecurityManager

Domain A

Permissions

Domain B

Permissions

I permessi in Java

- java.security.Permission Class

```
perm = new java.io.FilePermission("/tmp/abc","read");
```

java.security.AllPermission
java.security.SecurityPermission
java.security.UnresolvedPermission
java.awt.AWTPermission
java.io.FilePermission
java.io.SerializablePermission
java.lang.reflect.ReflectPermission
java.lang.RuntimePermission
java.net.NetPermission
java.net.SocketPermission
…

Classloader Hierarchy

Primordial ClassLoader

ClassLoader

SecureClassLoader

URLClassLoader

AppletClassLoader

Definizione dei privilegi

```
grant codeBase "http://www.l33t-hax0rz.com/*" {
  permission java.io.FilePermission("/tmp/*", "read,write");
}

grant codeBase "file://$JAVA_HOME/lib/ext/*" {
  permission java.security.AllPermission;
}

grant signedBy "trusted-company.com" {
  permission java.net.SocketPermission(…);
  permission java.io.FilePermission("/tmp/*", "read,write");
  …
}
```

Policy:

$JAVA_HOME/lib/security/java.policy
$USER_HOME/.java.policy

Trusted code

```
void fileWrite(String filename, String s) {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    FilePermission fp = new FilePermission(filename,“write”);
    sm.checkPermission(fp);
    /* … write s to file filename (native code) … */
  } else {
    throw new SecurityException();
  }
}
```

```
public static void main(…) {
  SecurityManager sm = System.getSecurityManager();
  FilePermission fp = new FilePermission(“/tmp/*”,“write,…”);
  sm.enablePrivilege(fp);
  UntrustedApplet.run();
}
```

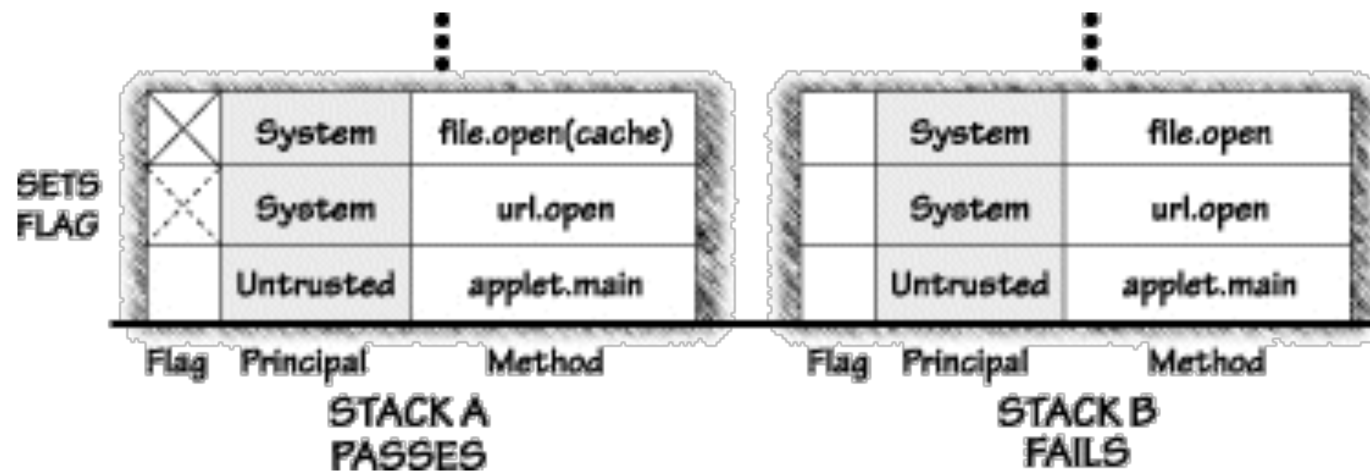
Applet scaricata http://www.l33t-hax0rz.com/

```
class UntrustedApplet {
  void run() {
    ...
    s.FileWrite(“/tmp/foo.txt”, “Hello!”);
    ...
    s.FileWrite(“/home/stevez/important.tex”, “kwijibo”);
    ...
  }
}
```

Stack inspection

- Record di attivazione sullo stack (stack frame nel gergo di Java) sono annotati con il loro livello di privilegio e i diritti di accesso.

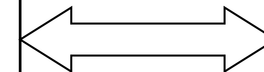
- Stack inspection: una ricerca sullo stack dei record di attivazione con l'obiettivo di determinare se il metodo in testa allo stack ha il diritto di fare una determinata operazione
  - **fail** se si trova un record di attivazione sullo stack che non ha i diritti di accesso
  - **ok** se tutti i record hanno il diritto di effettuare l'operazione


SETS FLAG
System
file.open(cache)
System
url.open
Untrusted
applet.main
Flag
Principal
Method
STACK A PASSES
System
file.open
System
url.open
Untrusted
applet.main
Flag
Principal
Method
STACK B FAILS

Esempio

Policy Database

```
main(…){
 fp = new FilePermission("/tmp/*","write,…");
 sm.enablePrivilege(fp);
 UntrustedApplet.run();
}
```



Esempio

Policy Database

```
main(…){
 fp = new FilePermission("/tmp/*","write,…");
 sm.enablePrivilege(fp);
 UntrustedApplet.run();
}
```

fp

Esempio

```
void run() {
 …
 s.FileWrite("/tmp/foo.txt", "Hello!");
 …
}
```

```
main(…){
 fp = new FilePermission("/tmp/*","write,…");
 sm.enablePrivilege(fp);
 UntrustedApplet.run();
}
```

fp

Policy Database

Esempio

```
void fileWrite("/tmp/foo.txt", "Hello!") {
 fp = new FilePermission("/tmp/foo.txt","write")
 sm.checkPermission(fp);
 /* … write s to file filename … */
```

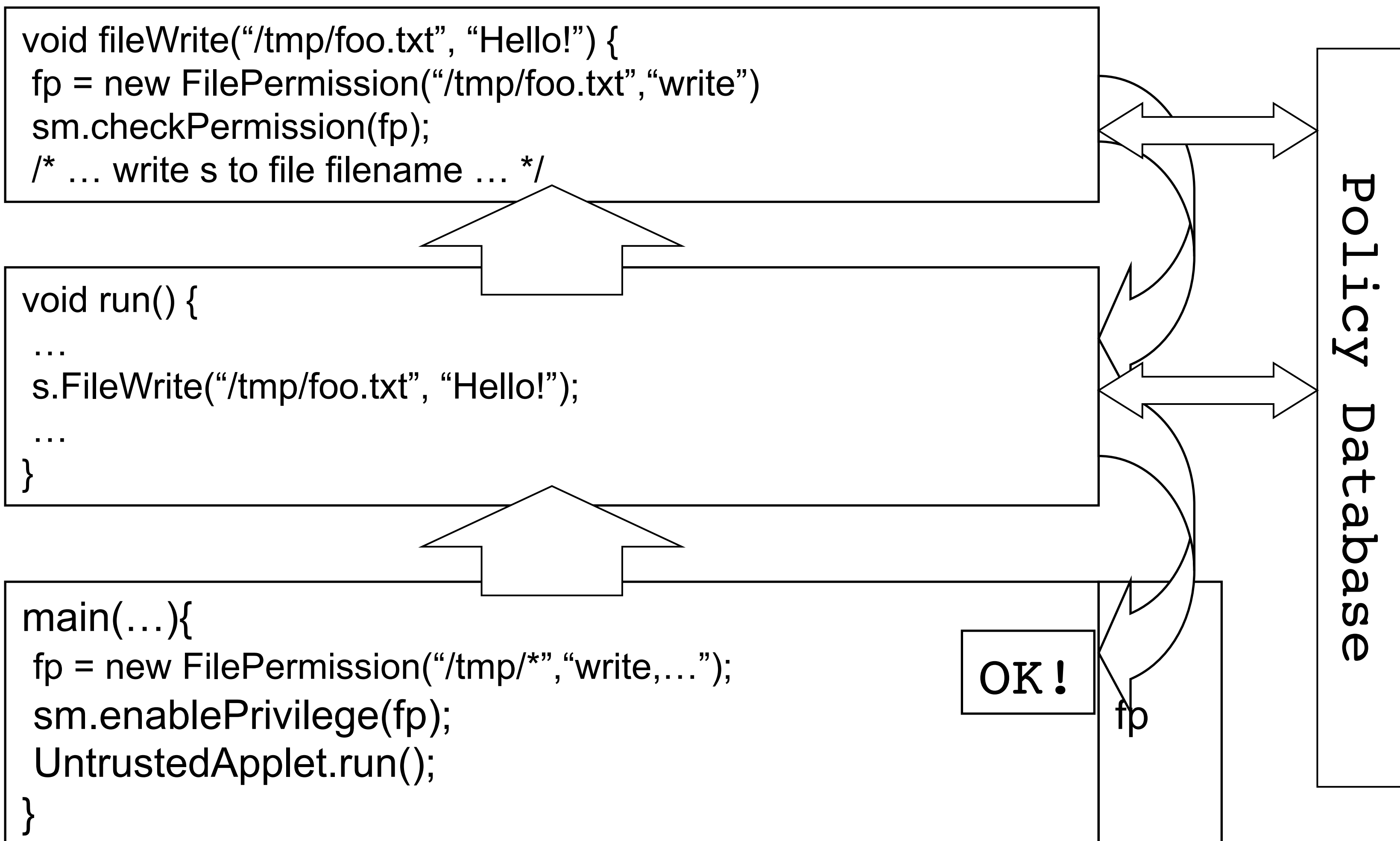
```
void run() {
 …
 s.FileWrite("/tmp/foo.txt", "Hello!");
 …
}
```

```
main(…){
 fp = new FilePermission("/tmp/*","write,…");
 sm.enablePrivilege(fp);
 UntrustedApplet.run();
}
```

fp

Policy Database

Esempio



Esempio