



AA 2015-2016

27. Classi e oggetti: implementazione

Dai sottoprogrammi...



- ✎ Un sottoprogramma (***astrazione procedurale***)
 - meccanismo linguistico che richiede di gestire dinamicamente **ambiente** e **memoria**
- ✎ La chiamata di sottoprogramma provoca la creazione di un ambiente e una memoria locale (record di attivazione) che esistono finché l'attivazione non restituisce il controllo al chiamante
- ✎ Ambiente locale
 - ✓ ambiente e memoria sono creati con la definizione della procedura
 - ✓ esistono solo per le diverse attivazioni di quella procedura

... alle classi



- ✎ L'aspetto essenziale dei linguaggi a oggetti consiste nella definizione di un meccanismo che permetta di creare ambiente e memoria al momento della “attivazione” di un oggetto (la creazione dell'oggetto)
 - nel quale gli ambienti e la memoria siano persistenti (sopravvivano alla attivazione)
 - una volta creati, siano accessibili e utilizzabili da chiunque possieda il loro meccanismo di accesso (“handle”)

Classi e loro istanziazione



- ✎ Tale meccanismo di astrazione linguistica è denominato **classe**
 - una classe è un costrutto linguistico che può avere parametri e come un normale sottoprogramma contiene un **blocco**
 - ✓ lista di **dichiarazioni**, e
 - ✓ lista di **comandi**
- ✎ L'istanziazione (attivazione) della classe avviene attraverso la chiamata del costruttore, ad esempio
 - `new(classe, parametri_attuali)` oppure
 - `new classe(parametri_attuali)`
 - che può occorrere in una qualunque espressione
 - con la quale si passano alla classe gli eventuali parametri attuali
 - che provoca la restituzione di un **oggetto**

Classi e istanziazione



- ✎ L'ambiente e la memoria locali dell'oggetto sono creati dalla valutazione delle **dichiarazioni**
 - le dichiarazioni di costanti e di variabili definiscono i **campi** dell'oggetto
 - ✓ se ci sono variabili, l'oggetto ha una memoria e quindi uno stato modificabile
 - le dichiarazioni di funzioni e procedure definiscono i **metodi** dell'oggetto
 - ✓ che vedono (e possono modificare) i campi dell'oggetto, per la normale semantica dei blocchi
- ✎ L'esecuzione della lista di **comandi** è l'inizializzazione dell'oggetto

Oggetti



- ✎ L'oggetto è la struttura (handle) che permette di accedere l'ambiente e la memoria locali creati permanentemente
 - attraverso l'accesso ai suoi metodi e campi
 - con l'operazione

Field(obj, id) (sintassi astratta)

obj.id (sintassi concreta)

- ✎ Nell'ambiente locale di ogni oggetto il nome speciale **this** denota l'oggetto medesimo

Oggetti e creazione dinamica di strutture dati



- La creazione di oggetti assomiglia molto (anche nella notazione sintattica) alla creazione dinamica di strutture dati tramite primitive linguistiche del tipo

`new(type_data)`

che provoca l'allocazione dinamica di un valore di tipo **`type_data`** e la restituzione di un puntatore a tale struttura

- Esempi: record in Pascal, struct in C

Strutture dati dinamiche



- ✎ Tale meccanismo prevede l'esistenza di una memoria a **heap**
- ✎ Strutture dati dinamiche: un caso particolare di oggetti, ma ...
 - hanno una semantica *ad hoc* non riconducibile a quella dei blocchi e delle procedure
 - non consentono la definizione di metodi
 - a volte la rappresentazione non è realizzata con campi separati
 - a volte non sono davvero permanenti
 - ✓ può esistere una (pericolosissima) operazione che permette di distruggere la struttura (**free**)



Ingredienti del paradigma OO

✎ Oggetti

- meccanismo per incapsulare dati e operazioni

✎ Ereditarietà

- riuso del codice

✎ Polimorfismo

- principio di sostituzione

✎ Dynamic binding

- legame dinamico tra il nome di un metodo e il codice effettivo che deve essere eseguito



Implementazione: var. di istanza

- ✎ Soluzione: ambiente locale statico che contiene i binding delle variabili di istanza
 - con associato il descrittore di tipo

```
class A {  
    int a1;  
    int a2;  
}  
A obj = new A(4,5)
```

Descrittore

CLASS A	
a1	4
a2	5

E
N
V



E la ereditarietà?

```
class A {  
    int a1;  
    int a2;  
}
```

```
class B extends A {  
    int a3  
}
```

CLASS B	
a1	
a2	
a3	

Soluzione: i campi ereditati dall'oggetto vengono inseriti all'inizio nell'ambiente locale

Oggetti: ambiente locale statico



- ✎ L'utilizzo di un ambiente locale statico permette di implementare facilmente la persistenza dei valori
 - gestione della ereditarietà (singola) è immediata
 - gestione dello shadowing (variabili di istanza con lo stesso nome usata nella sottoclasse) è immediata
- ✎ Se il linguaggio prevede meccanismi di controllo statico si può facilmente implementare un accesso diretto: **indirizzo di base + offset**

E i metodi?

- ✎ Soluzione: associare un puntatore alla tabella (*ambiente locale statico*) che contiene il binding dei metodi e il descrittore della classe

```
class A {  
  int a1;  
  int a2;  
  int m1 ...;  
  void m2 ...;  
}
```

a1	4
a2	5



CLASS A	
m1	code
m2	code

```
A obj = new A(4,5)
```



Analisi

- ✎ Dal puntatore memorizzato nell'ambiente locale delle variabili di istanza si accede al descrittore della classe
- ✎ Dal descrittore della classe si ottengono le informazioni relative all'allocazione dei metodi
- ✎ Implementazione dell'accesso diretto ai metodi con il solito meccanismo (indirizzo di base + offset)

Implementazione dei metodi



- ✎ Un metodo è eseguito come una funzione (implementazione standard: AR sullo stack con variabili locali, parametri, ecc.)
- ✎ **Importante:** il metodo deve poter accedere alle variabili di istanza dell'oggetto sul quale è invocato (che non è noto al momento della compilazione)
- ✎ **L'oggetto è un parametro implicito:** quando un metodo è invocato, gli viene passato anche un puntatore all'oggetto sul quale viene invocato; durante l'esecuzione del metodo il puntatore è il **this** del metodo

Ereditarietà

🐼 Soluzione1 (Smalltalk)

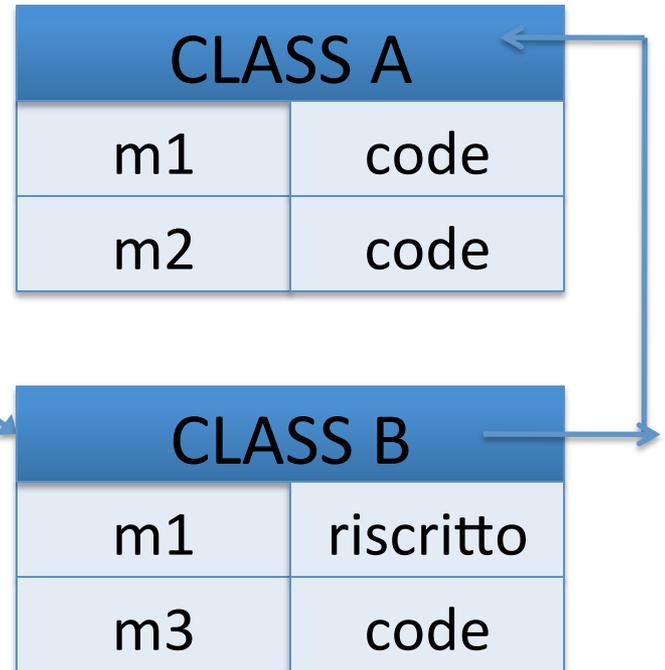
- o lista di tabelle

```
class A {
  int a1, a2;
  void m1 ...;
  void m2 ...;
}
```

CLASS A	
a1	
a2	
a3	

```
class B extends A
  int a3;
  void m1 ...;
  void m3 ...;
}
```

```
new B(...)
```



Ereditarietà

🐼 Soluzione 2 (C++)

- sharing strutturale

```
class A {
  int a1, a2;
  void m1 ...;
  void m2 ...;
}
```

CLASS A	
a1	
a2	
a3	

CLASS A	
m1	code
m2	code

CLASS B	
m1	riscritto
m2	code
m3	code

```
class B extends A
  int a3;
  void m1 ...;
  void m3 ...;
}
```

`new B(...)`

Analisi



- ✎ Liste di tabelle dei metodi (Smalltalk):
l'operazione di dispatching dei metodi viene risolta con una visita alla lista (overhead a run time)
- ✎ Sharing strutturale (C++): l'operazione di dispatching dei metodi si risolve staticamente andando a determinare gli offset nelle tabelle (**vtable** in C++ [virtual function table])

Discussione: Smalltalk



- ✉ Smalltalk (ma anche JavaScript) non prevedono un meccanismo per il controllo statico dei tipi
 - l'invocazione di dispatch del metodo `obj.meth(pars)` dipende dal flusso di esecuzione
 - ogni classe ha il proprio meccanismo di memorizzazione dei metodi nelle tabelle

Discussione: C++

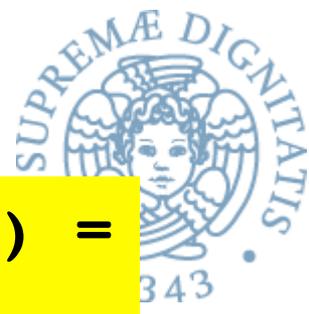


- ✎ C++ prevede un controllo dei tipi statico degli oggetti
 - offset dei campi degli oggetti (`offset_data`), la struttura delle vtable è condivisa nella gerarchia di ereditarietà
 - offset dei dati e dei metodi sono noti a tempo di compilazione
- ✎ Il dispatching “**`obj.mth(pars)`**”
 - `obj->mth(pars)`** nella notazione C++ viene pertanto compilato nel codice
 - `*(obj->vptr[0])(obj, pars)`**
 - assumendo che **`mth`** sia il primo metodo della vtable
- ✎ Si noti il passaggio dell'informazione relativa all'oggetto corrente

Compilazione separata



- ✎ Compilazione separata di classi (**Java**): la compilazione di una classe produce un codice che la macchina astratta del linguaggio carica dinamicamente (**class loading**) quando il programma in esecuzione effettua un riferimento alla classe
- ✎ In presenza di compilazione separata gli offset non possono essere calcolati staticamente a causa di possibili modifiche alla struttura delle classi



```
class A {  
  :  
  void m1() {...}  
  void m2() {...}  
}
```



**access(A,m1()) =
offset1**

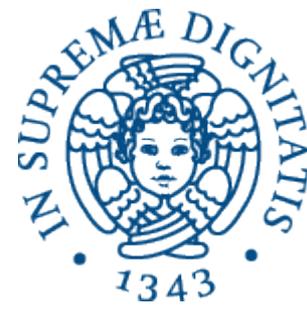


Raffinamento della struttura di A

```
class A {  
  :  
  void ma() {...}  
  void mb() {...}  
  void m1() {...}  
  void m2() {...}  
}
```

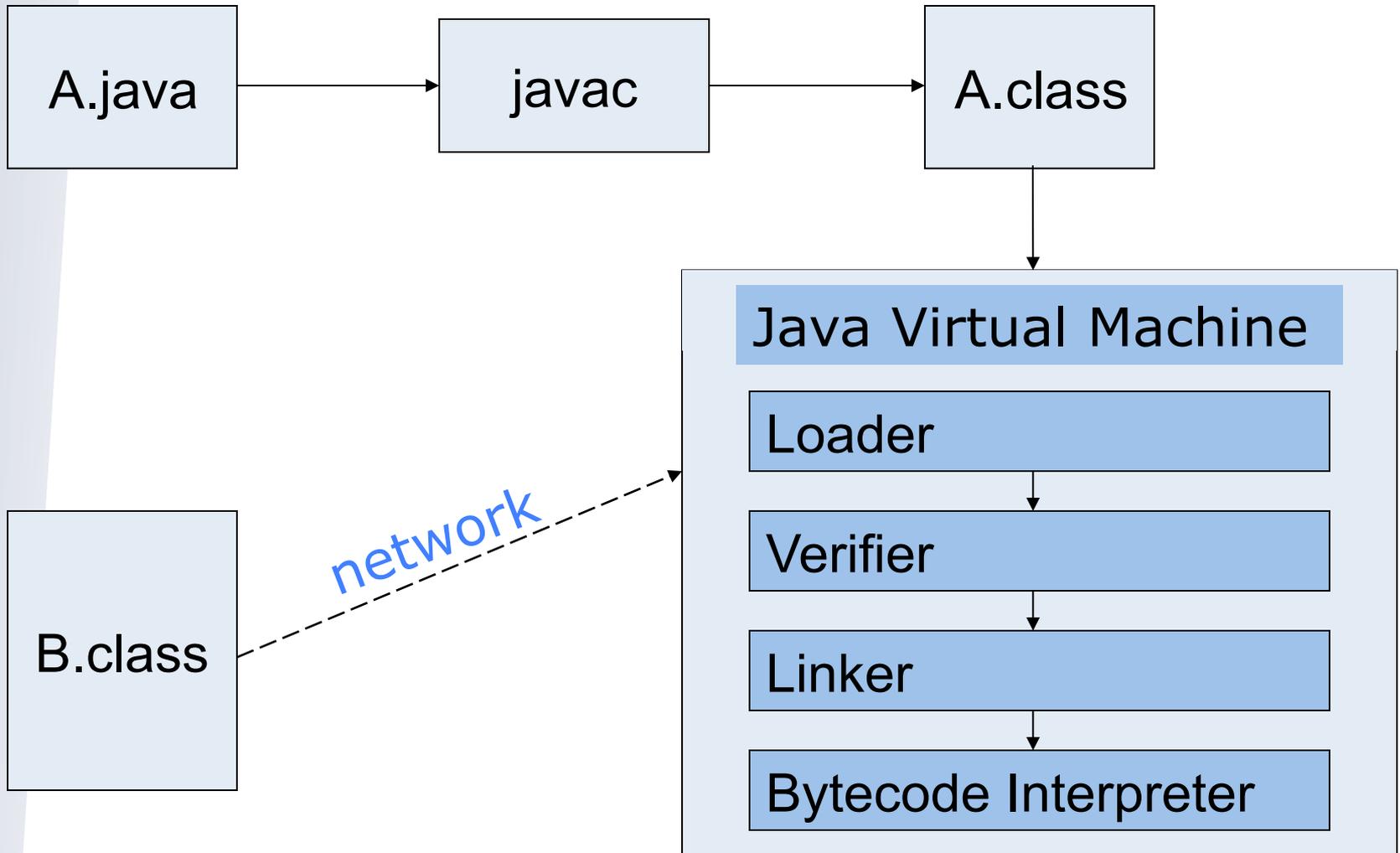


**access(A,m1()) =
offset3
[!= offset1]**

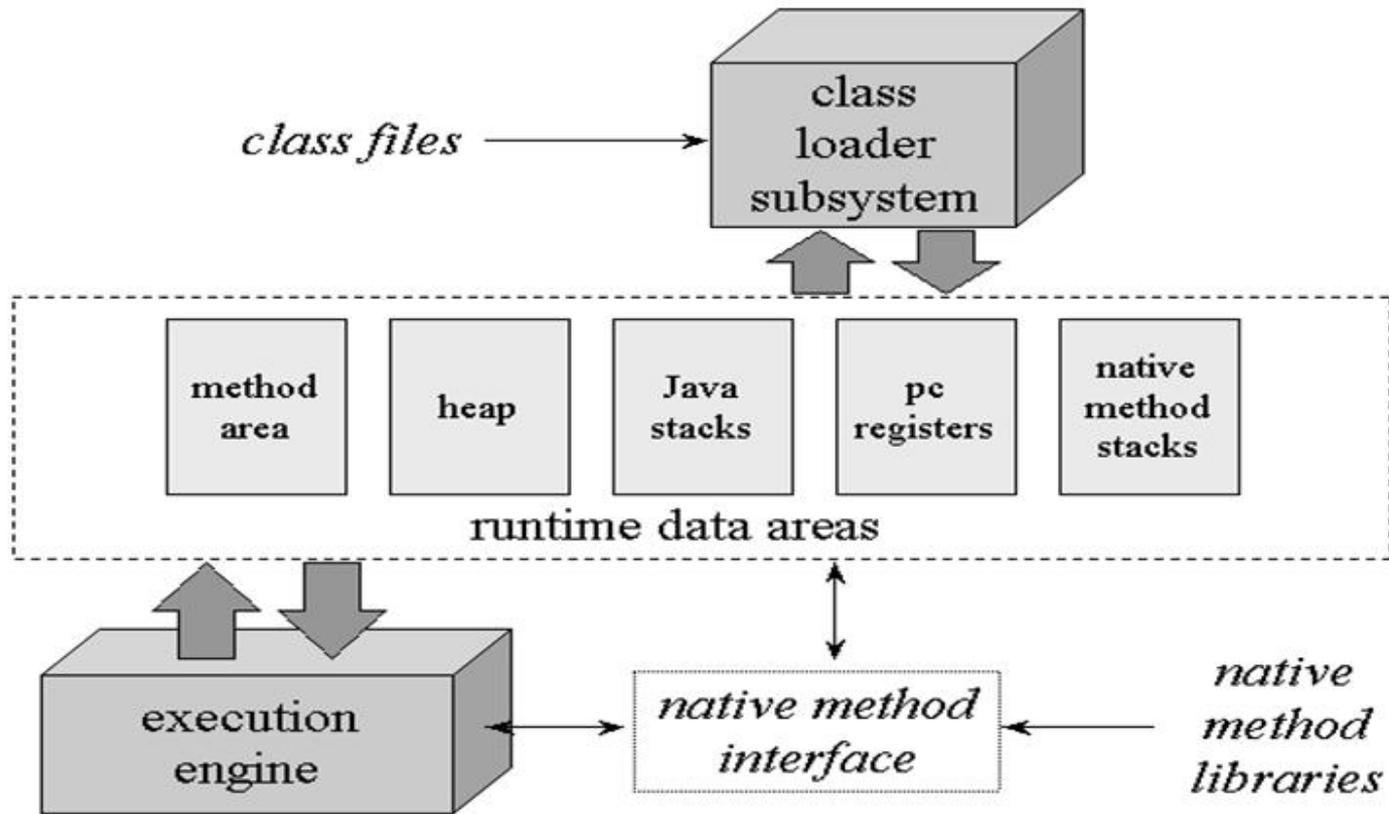


Come lo risolve Java?

JVM: visione di insieme



JVM





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>

.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 all'informazione**



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;}  
}
```

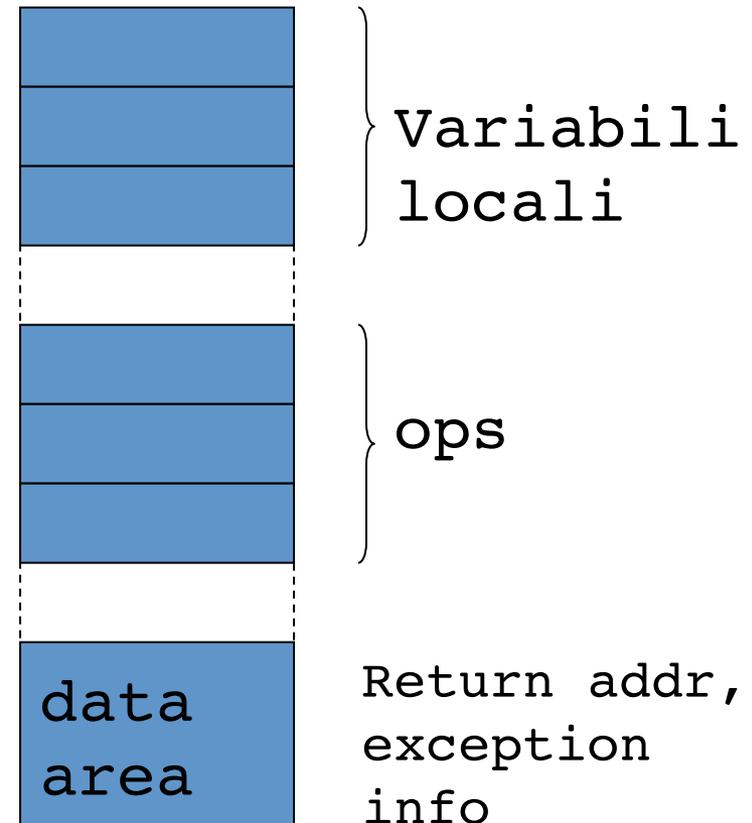
@ Bytecode

```
Method void f(int)  
    aload 0 ; object ref this  
    iload 1 ; int val  
    iconst 1  
    iadd ; add val +1  
    putfield #4 <Field int i>  
    return
```



riferimento alla const pool

JVM Activation Record





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
```

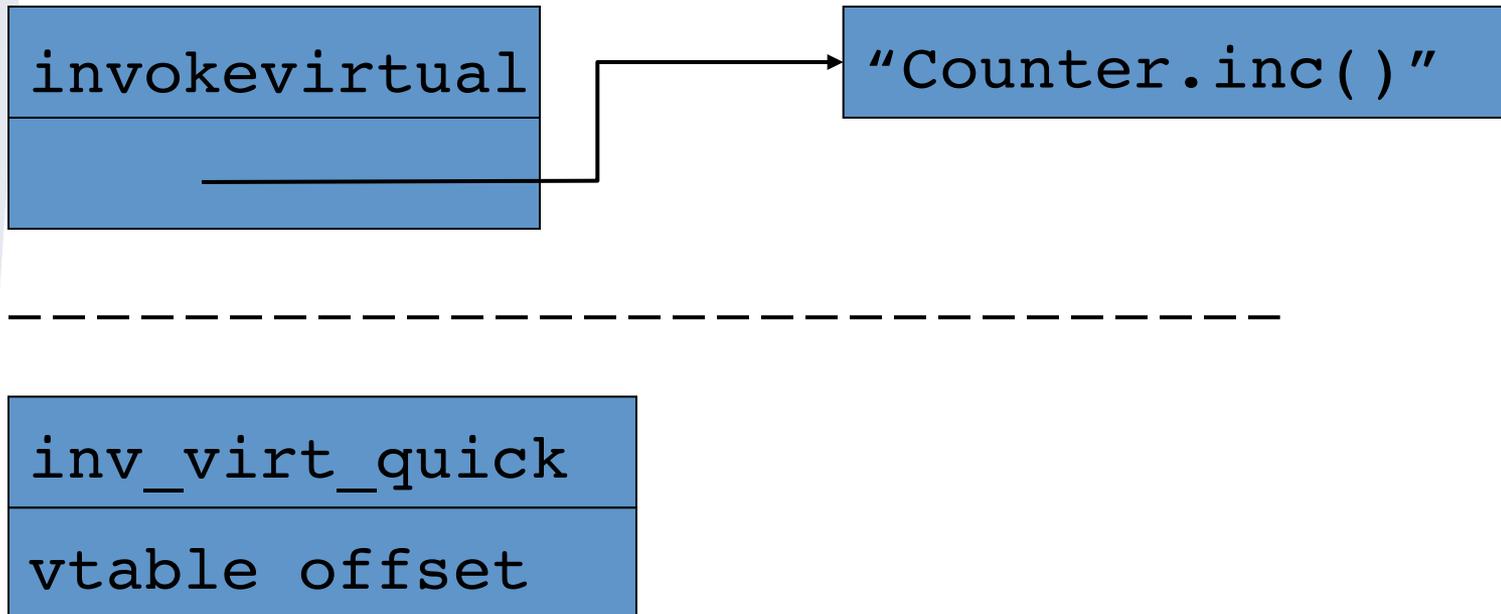
```
28:  return
```



Bytecode : invokevirtual

Bytecode

Constant pool



- ☞ 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() { .. }  
    :  
}
```

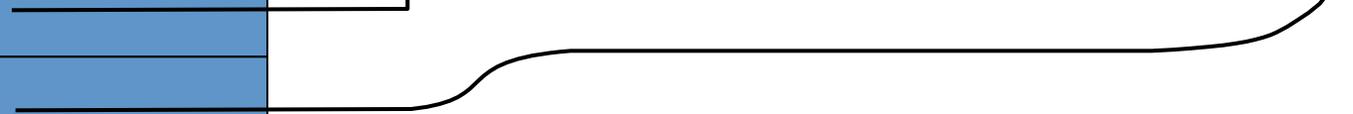
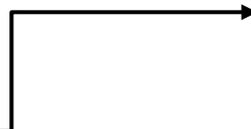
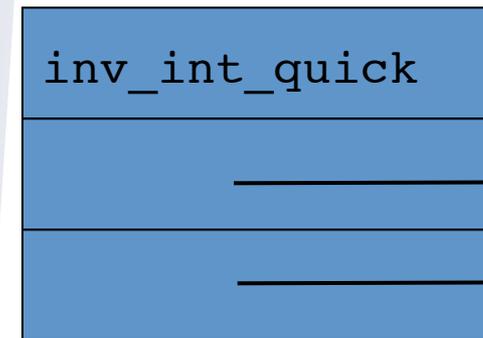
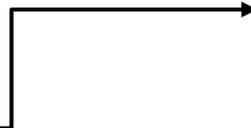
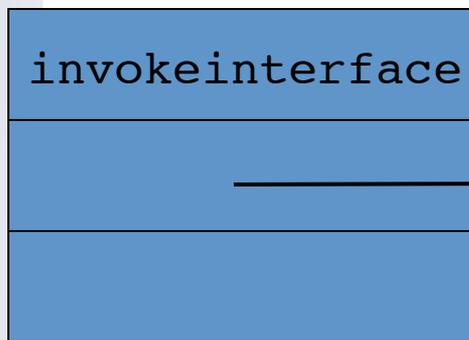
```
public class B implements I {  
    :  
    void m() { .. }  
    void foo() { .. }  
    :  
}
```

Bytecode: invokeinterface



Bytecode

Constant pool



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 useAltoHashing;  
    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.**

Ereditarietà multipla



```
class A { int m(); }
class B { int m(); }
class C extends A, B { }
// quale metodo si eredita??
```

```
class A { int x; }
class B1 extends A { ... }
class B2 extends A { ... }
class C extends B1, B2 { ... }
// "diamond of death"
```

Ereditarietà multipla

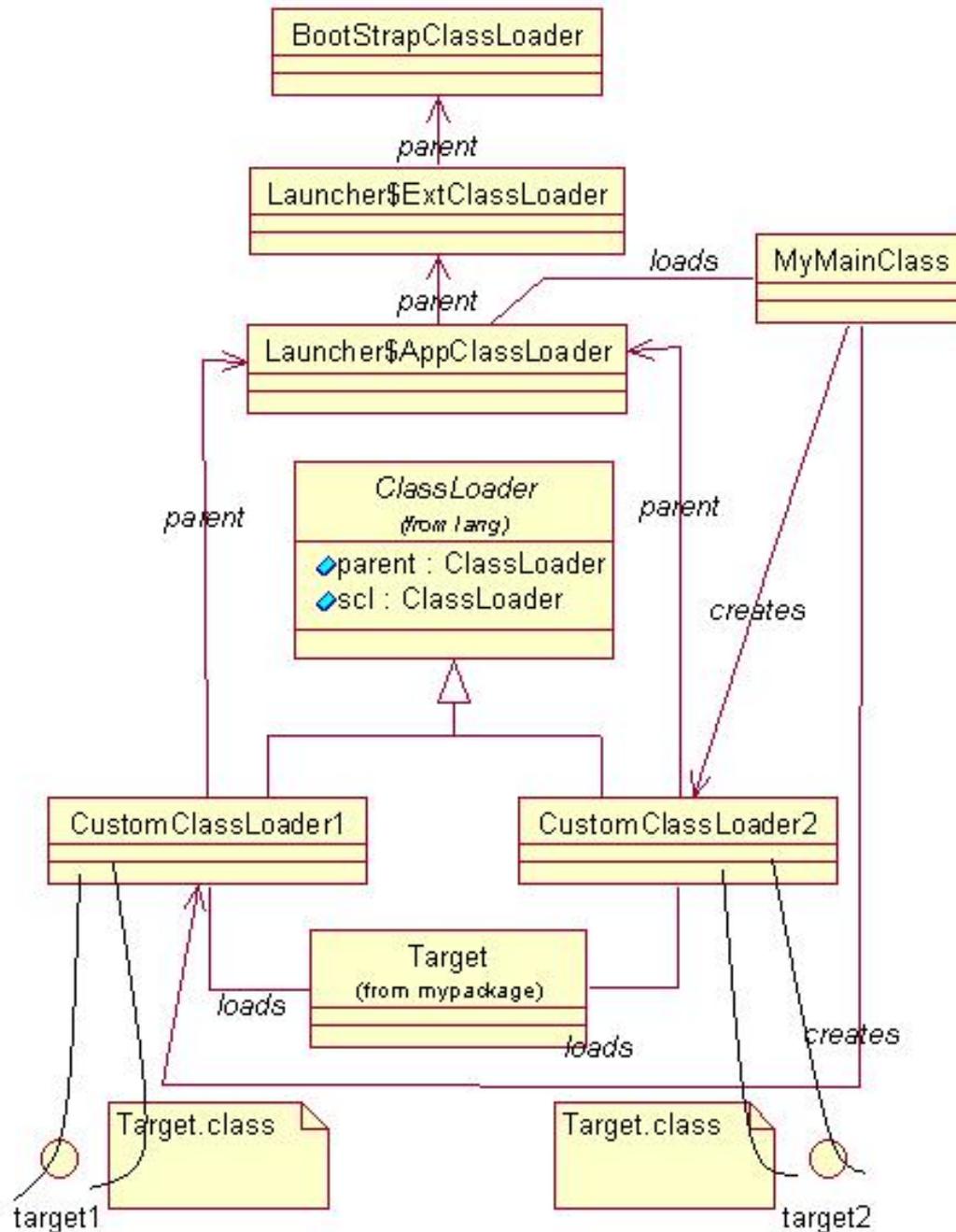


- ✎ Complicazione della compilazione
- ✎ Complicazione delle struttura a run-time
- ✎ Noi non lo trattiamo (alcuni dettagli nel libro di Gabbrielli e Martini)

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



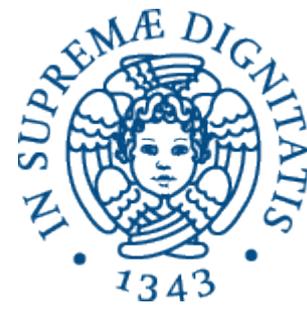
```
class A {  
    String name;  
    A(String s) {  
        name = s;  
    }  
}
```

```
<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)
```

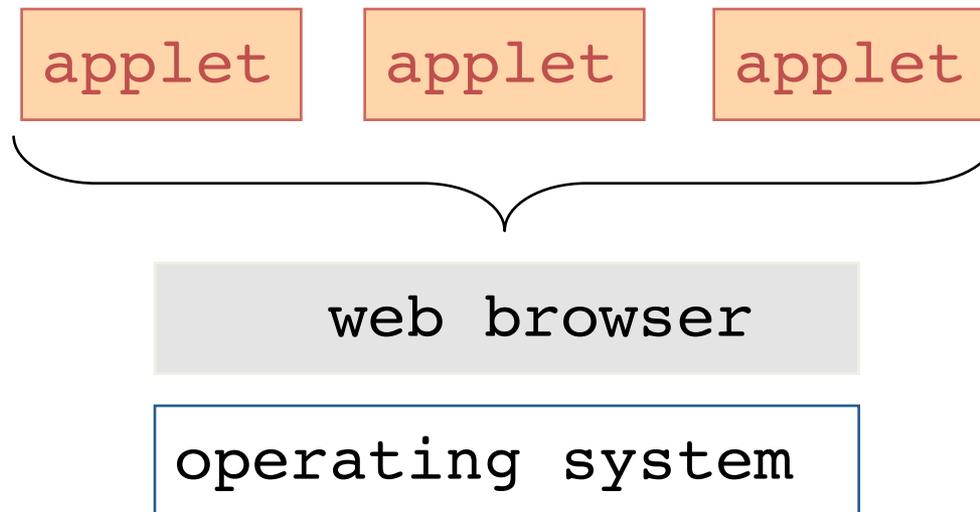


Java stack inspection

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



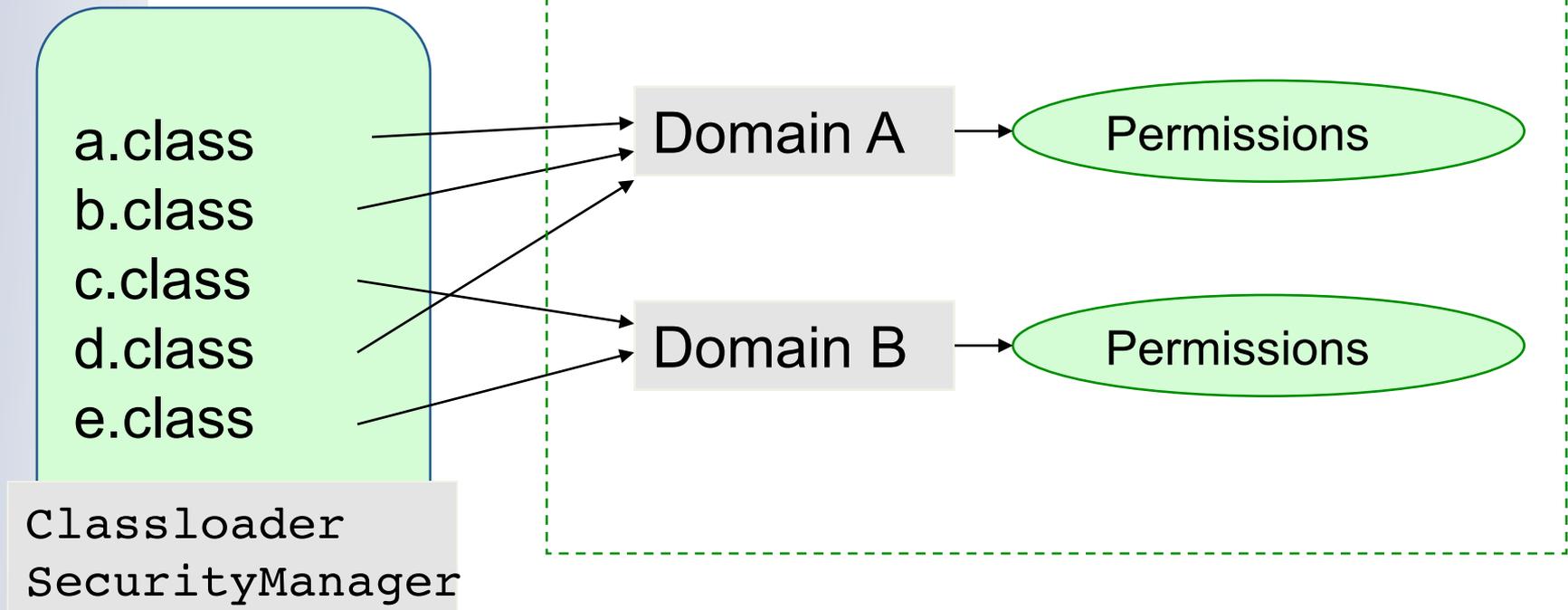
- ✎ Fornitori di servizio hanno livelli di sicurezza differenti (classico dei SO)
 - `www.l33t-hax0rs.com` vs. www.java.sun.com (ci fidiamo?)
 - untrusted code vs trusted code
- ✎ Trusted code può invocare untrusted code
 - e.g. invocare una applet per visionare dei dati
- ✎ Untrusted code può invocare 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





I permessi in Java

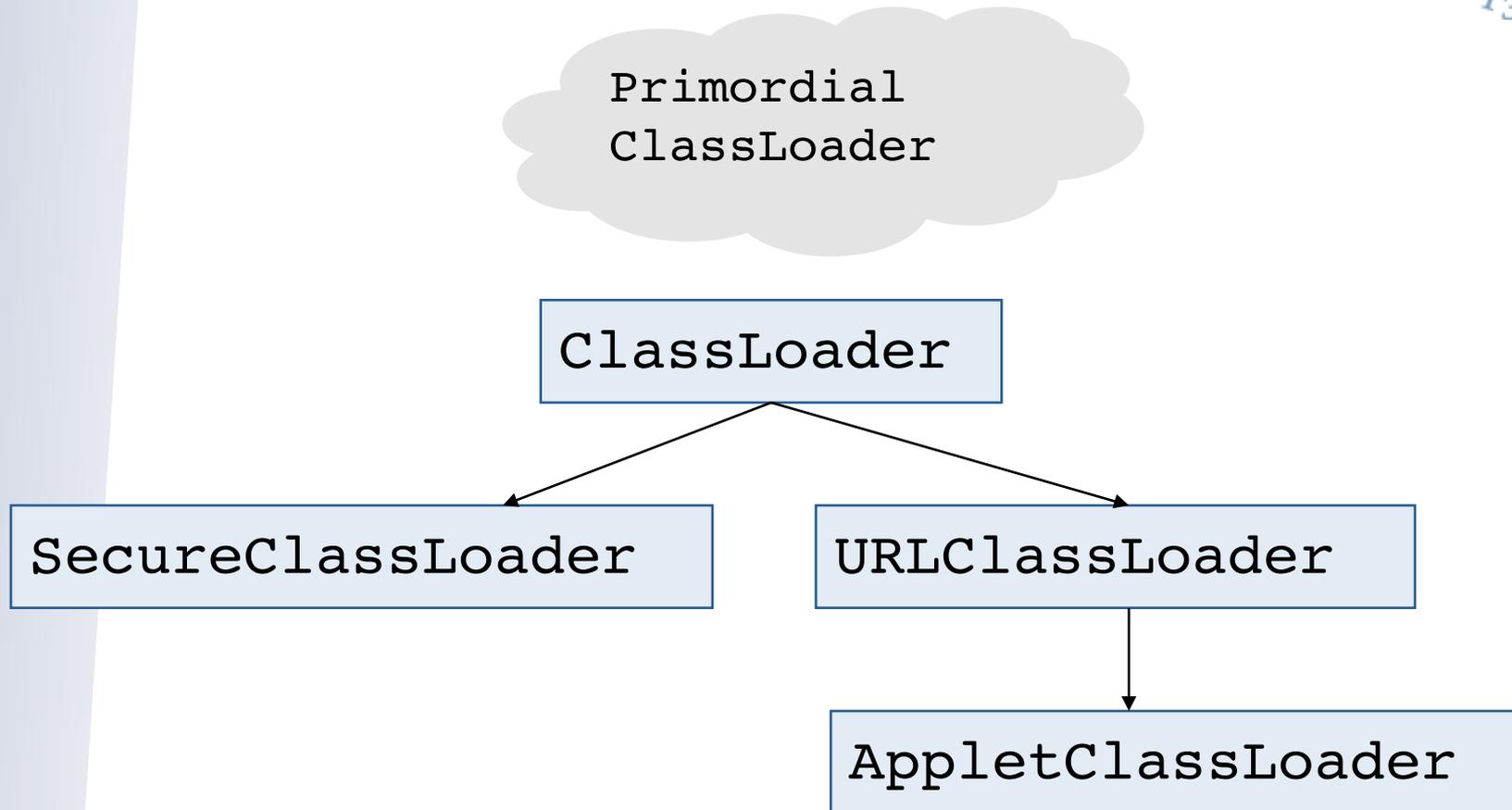
java.security.Permission Class

```
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



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

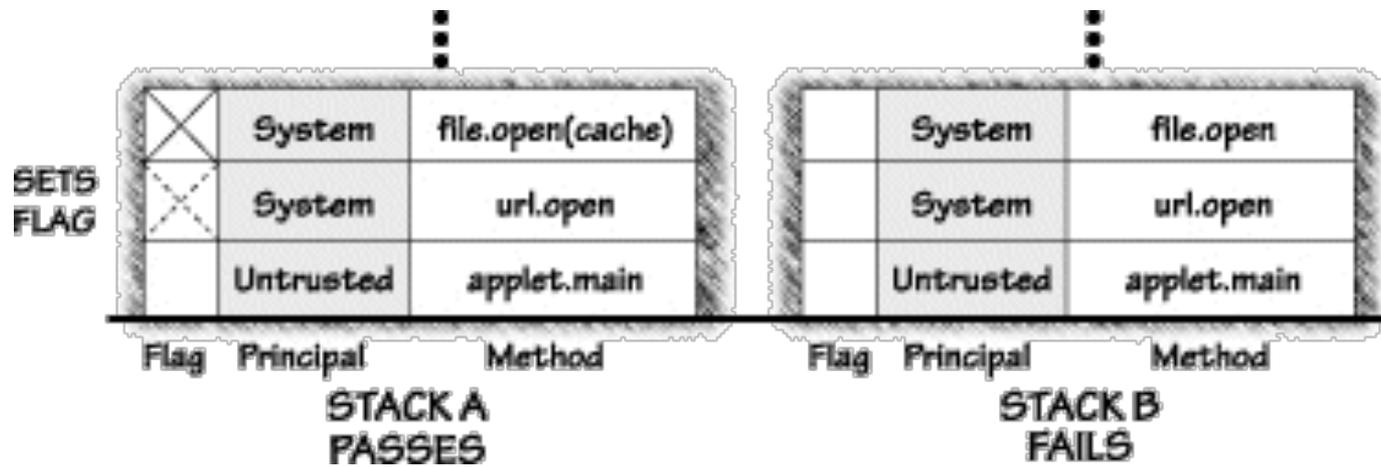
<http://www.l33t-hax0rz.com/>

```
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



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



Policy Database

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

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

fp



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

```
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();  
}
```

OK!

fp

Policy Database



Esempio

```
void fileWrite("../important.txt", "kwijibo") {  
    fp = new FilePermission("important.txt", "write");  
  
    sm.checkPermission(fp);  
}
```

```
void run() {  
    ...  
    s.FileWrite("/home/stevez/important.tex", "kwijibo");  
    ...  
}
```

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

fp

Fail

Policy Database