Modello e Architettura di Sicurezza in Java

Indice

- Costrutti di sicurezza a livello di linguaggio
  - "type safety" di Java
- Evoluzione dei modelli di sicurezza in Java
- Architettura di sicurezza nel JDK 1.2
- Supporto crittografico di Java

Costrutti di “sicurezza” a livello di linguaggio (1.)

- Utilizzo controllato delle variabili in memoria:
  - ogni entità ha un livello di protezione associato (private, default, protected, public)
- Accesso controllato alle locazioni di memoria
- Utilizzo delle variabili solo una volta inizializzate
- Proibizione di casting arbitrario tra oggetti

Eccezione: serializzazione (introduzione della label *transient*)

Costrutti di “sicurezza” a livello di linguaggio (2.)

Esempio:

Classe corretta (frammento):

```
public class CreditCard {
    private String acctNo;
    ……...
}
```

Classe “falsificata” (frammento):

```
public class CreditCardSnoop {
    public String acctNo;
}
```

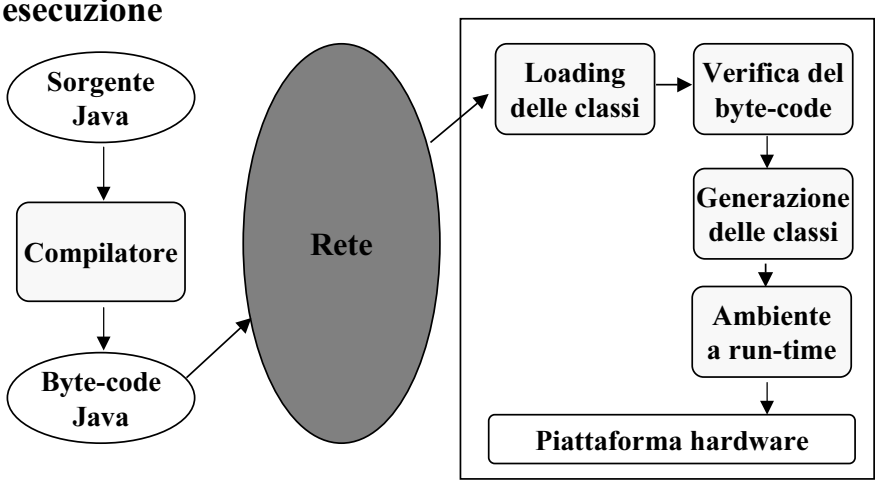
Questo codice dà errore in compilazione:

```
CreditCard cc =
Wallet.getCreditCard();
CreditCardSnoop snoop =
(CreditCardSnoop) cc;
System.out.println(“Il numero
di carta di credito è”
+snoop.acctNo);
```

Controlli di "Type Safety"

Quando vengono effettuati i controlli?

- **a tempo di compilazione**;
- **a tempo di caricamento delle classi;**
- **a tempo di esecuzione**



Controlli a Tempo di Compilazione

Il compilatore ***controlla*** che:

- le variabili in memoria siano accessibili solo quando permesso;
- le locazioni di memoria non siano accedute arbitrariamente;
- l'utilizzo delle variabili avvenga ad inizializzazione avvenuta;

non controlla tutti i casting tra oggetti.

Esempio di casting verificato a tempo di compilazione:

```
Vector v = new Vector();
String s = (String) v
```

⟺ Vector e String sono notoriamente due oggetti non correlati tra loro

Controlli a Tempo di Caricamento (1.)

Le classi caricate vengono sottoposte ad un processo di verifica (**bytecode verification**) che esamina la correttezza dei bytecode (il bytecode viene letto dalla JVM sotto forma di file con estensione *class*), cioè se una certa sequenza di bytecode rappresenta un insieme legale di istruzioni Java.

Bytecode Verifier = parte interna della Java Virtual machine senza interfaccia. Nè i programmatori né gli utenti possono interagire con esso o accedere a sue parti.

Controlli a Tempo di Caricamento (2.)

In particolare, il **Bytecode Verifier** esamina se:

- il file .class ha un formato corretto;
- ogni classe ha una superclass da cui eredita;
- esistono conversioni illegali tra tipi primitivi (e.g., *int* in *Object*);

Se la verifica termina con successo, il codice in questione si può ritenere “type safe” fino a questo punto

Verifica Ritardata del Bytecode

Non tutti i bytecode vengono immediatamente verificati:

```
CreditCard cc = getCreditCard();
try {
    Wallet.makePurchase(CC);
} catch (IllegalArgumentException iae) {
    System.out.println("Non si può processare il conto corrente" +
    cc.acctNo);
```

=> il controllo su acctNo viene effettuato solo se l'eccezione IllegalArgumentException viene lanciata

Controlli a Tempo di Esecuzione

Vengono controllati:

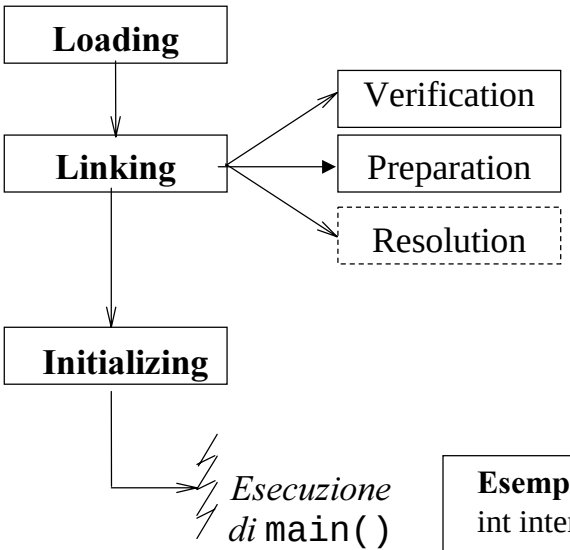
- estremi degli array
- casting tra oggetti

Esempio:

```
void initArray(int a[], int nItems) {
    for (int i = 0; i<nItems; i++ {
            a[i] = 0;
        }
    }
```

Fallimento ⇒ ArrayIndexBoundException

Java Virtual Machine



caricamento in memoria del file *.class*

verifica byte-code

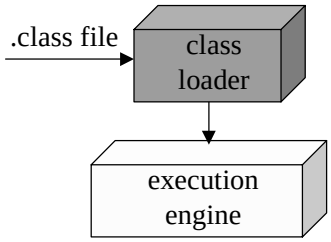
allocazione spazio in memoria

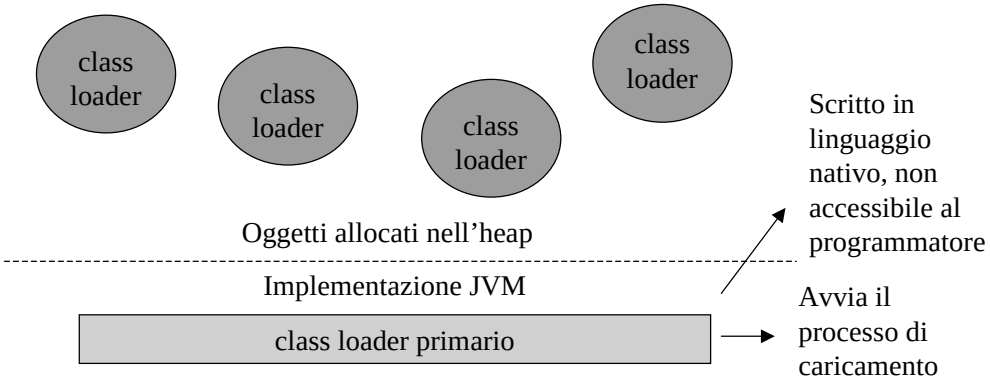
risoluzione di tutti i riferimenti simbolici ad altre classi

inizializzazione di tutte le variabili statiche allocate

Esempio:
int intero = 5
preparation: memoria per l'int
inizializing: il valore di int viene modificato in 5

Il Class Loader di Java (1.)





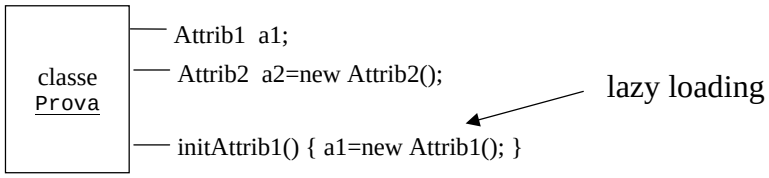
Il Class Loader di Java (2.)

Caratteristiche:

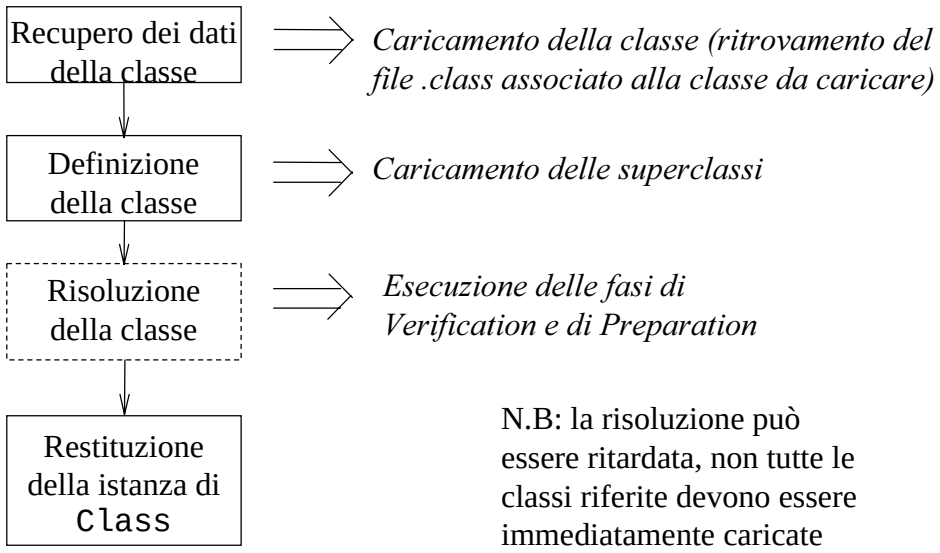
- **programmabilità**
- **"lazy" loading**; le classi sono caricate "on demand" e possibilmente all'ultimo momento

Esempio:

Prova p = new Prova();

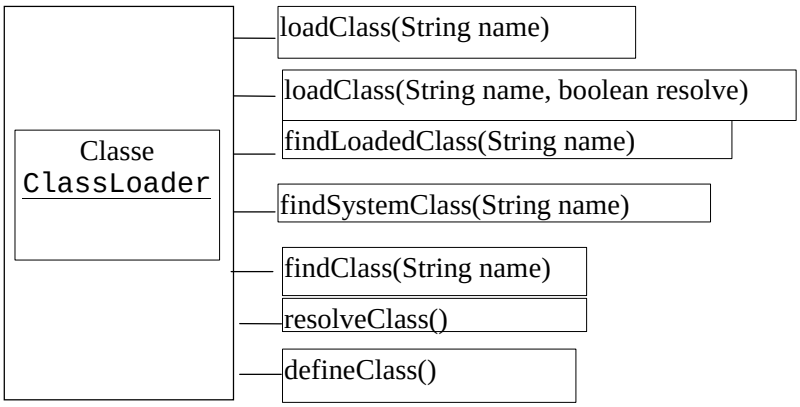


Il Class Loader di Java (3.)



N.B: la risoluzione può essere ritardata, non tutte le classi riferite devono essere immediatamente caricate

Struttura della classe Class Loader



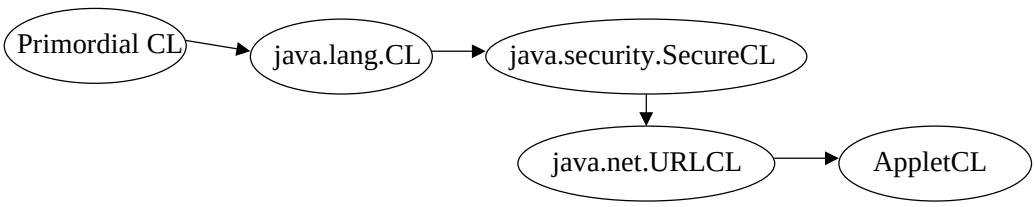
Implementazione di default del metodo loadClass:

- chiamata al metodo findLoadedClass()
- chiamata al findSystemClass()
- chiamata al findClass()

Tipi di Class Loader

- *class loader interno*; in JDK 1.2 usato solo per caricare le classi di Java;
- *secure class loader*; in JDK 1.2 permette di implementare politiche di sicurezza più granulari;
- *applet class loader*; ogni browser implementa un proprio class loader;
- *URL class loader*; permette di caricare classi da un insieme di URL distinte

I programmatori possono definire class loader propri purchè tutti estendano i tipi predefiniti



Relazione tra Class Loader: Delega

• La relazione che sussiste tra le istanze di class loader è una relazione di delega => quando un'istanza di class loader carica una classe puo':

- direttamente gestire il caricamento
- delegare un altro class loader, tipicamente il class loader padre

La relazione di delega si crea al momento della creazione delle istanze di class loader:

• protected ClassLoader(ClassLoader parent)

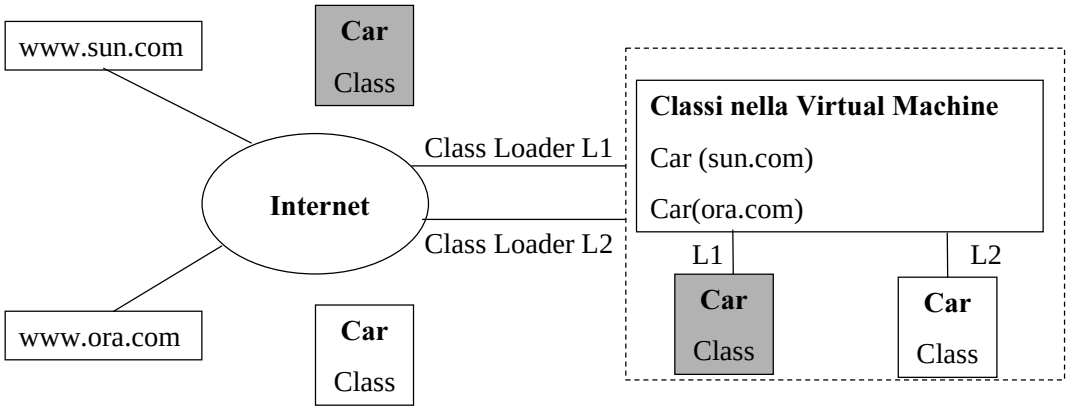
• protected ClassLoader()

Il padre di default è determinato chiamando il metodo getParent() oppure getSystemClassLoader e tipicamente è il class loader utilizzato per avviare l'applicazione

Class Loader e Sicurezza

Il Class Loader gioca un ruolo fondamentale

- crea spazi di nomi separati (garantisce type safety e visibilità protetta delle classi);
- si coordina con il componente che implementa le politiche di sicurezza per estrarre i permessi associati alla classe



Class Loader e Name-Spaces

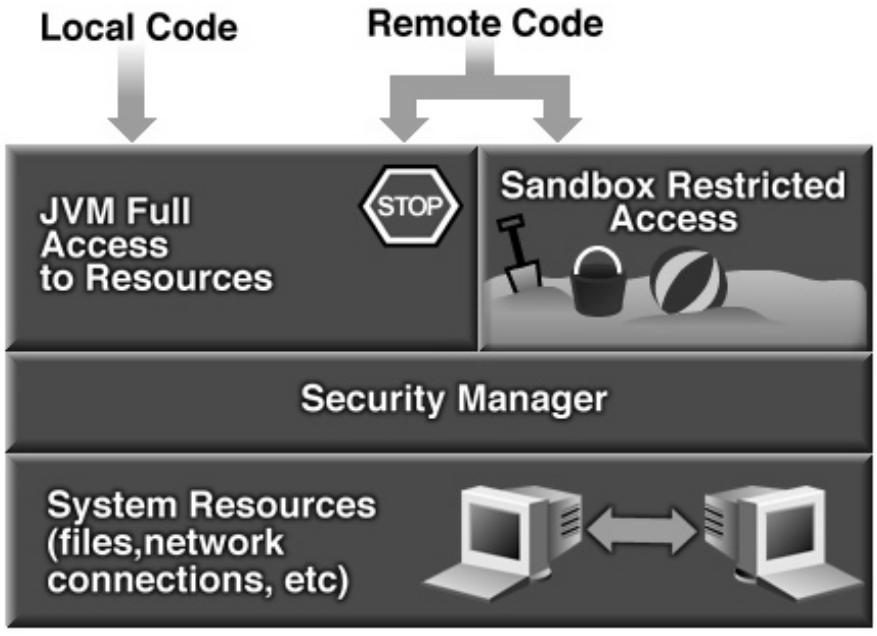
```
class C {
      void f() {
          D c = new D();
          …..}
}
```

```
MyClassLoader cl= new MyClassLoader();
Class provaClass = cl.loadClass("/foo/bar/C");
```

Sia cl l'istanza di class loader che carica la classe C (*defining* class loader)
Tipo di classe = <C, cl>
Due tipi di classi sono uguali se il tipo della classe è uguale e sono uguali i defining class loader.

Per ogni classe la JVM tiene traccia di quale istanza di class loader l'ha caricata.
Quando la classe caricata (C) riferisce altre classi, la JVM incarica il defining class loader di C di caricare le classi riferite =>
spazio dei nomi: insieme di nomi di classi caricate da un'istanza di class loader (C e D appartengono allo stesso spazio di nomi).
La stessa istanza di class loader non puo' caricare due classi con nomi uguali.

Modello di Sicurezza in JDK™ 1.0 (1.)



Modello di Sicurezza in JDK™ 1.0 (2.)

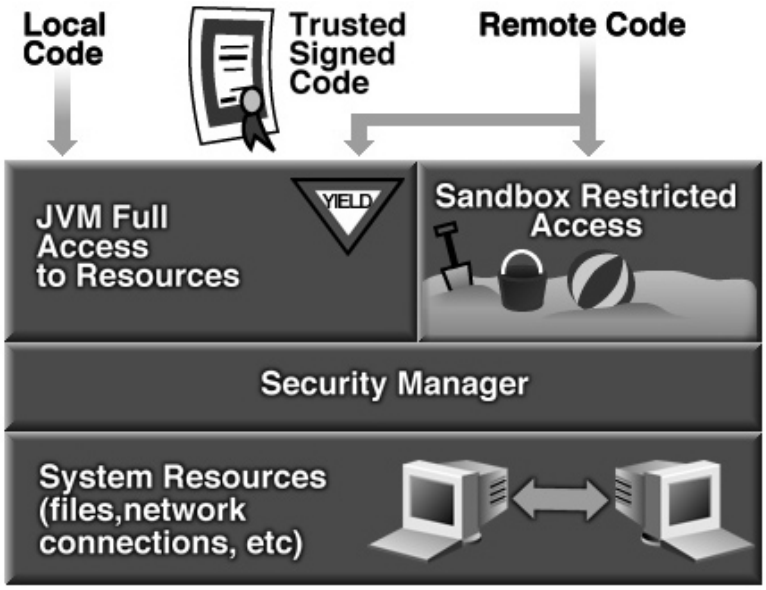
Vantaggi:

- la sandbox protegge l'accesso a tutte le risorse di sistema
- i programmatori di applicazioni (non di applet) **possono scrivere** un proprio Security Manager per "aprire" la sandbox

Limitazioni:

- modello troppo restrittivo
- i programmatori di applicazioni (non di applet) **devono scrivere** un proprio Security Manager per "aprire" la sandbox
- ad ogni politica diversa corrisponde una nuova versione del SecurityManager <=> assenza di separazione tra politiche e meccanismo

Modello di Sicurezza in JDK™ 1.1 (1.)



Modello di Sicurezza in JDK™ 1.1 (2.)

Vantaggi:

- la firma del codice assicura:
  - autenticazione
  - integrità

Limitazioni:

- applicazioni locali non sono sottoposte ad alcun controllo di accesso
- tutte le classi presenti nel CLASSPATH sono considerate fidate

Modello di Sicurezza in JDK™ 1.2 (1.)

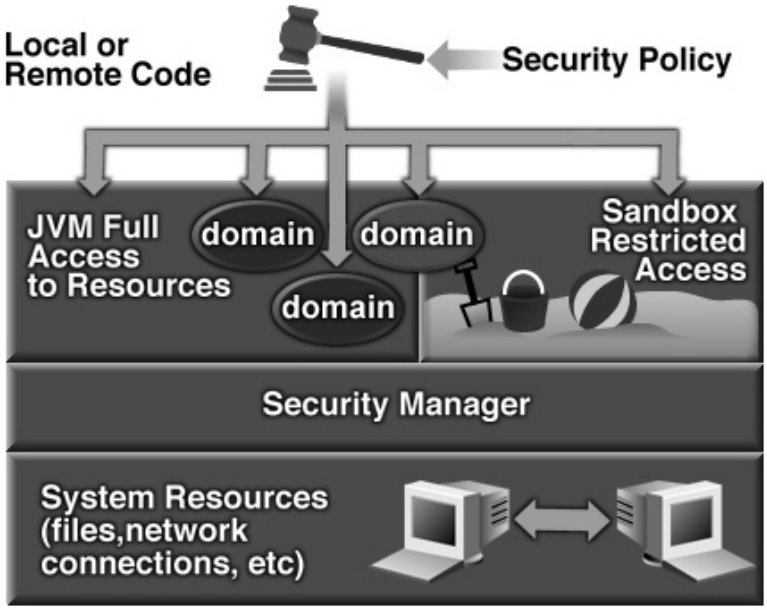
Obiettivi:

- estensione dei controlli di sicurezza sia alle applet sia alle applicazioni (esterne e locali)
- configurabilità semplificata delle politiche di sicurezza
- controlli di accesso granulari

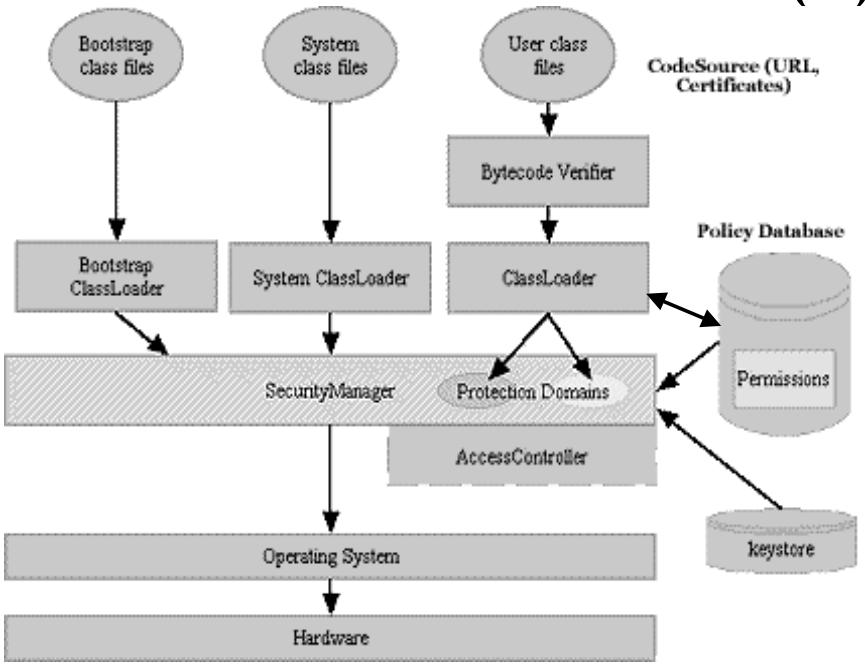
Elementi caratterizzanti la nuova architettura:

- separazione tra politiche e meccanismi di sicurezza
- definizione di permessi di accesso
- controllo d'accesso granulare per tutte le classi

Modello di Sicurezza in JDK™ 1.2 (2.)



Modello di Sicurezza in JDK™ 1.2 (3.)



Politica di Sicurezza (1.)

• Politica di sicurezza: matrice di controllo dell'accesso

Codice	Permessi
Li Gong applet, applet firmate	read, write /tmp and home/gong

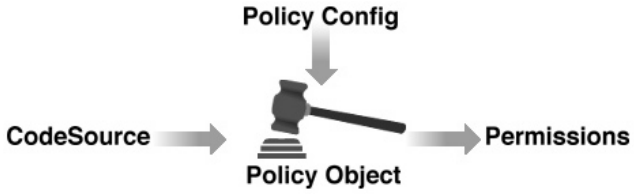
Esempio di Politica di Sicurezza (rappresentazione di default):

grant *signedBy* "*", *CodeBase* "http://java.sun.com/people/gong/"

```
{
    permission java.io.FilePermission "read, write", "tmp/*";
}
```


Politica di Sicurezza (2.)

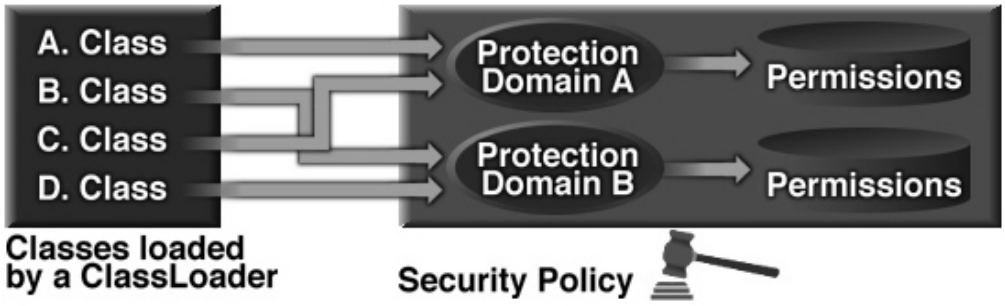
- **CodeSource** (chi/dove): località di provenienza del codice e insieme di certificati utilizzati per la firma del codice
- **Permissions:** permessi attribuiti al codice
- abstract class **Policy**:
  - public static Policy getPolicy()
  - public static void setPolicy(Policy policy)
  - public abstract Permissions getPermissions(CodeSource codesource)
  - public abstract void refresh()



Secure Class Loader e Protection Domain

Protected final Class

defineClass(String name, byte [] b, int off, int len, CodeSource cs)



Esempio di Secure Class Loader (1.)

```
public class AgentClassLoader extends SecureClassLoader
……..
public AgentClassLoader() {}
 public AgentClassLoader( Environment env, String agentClass, AgentID agentID )
  {  …...
     codeSource = new CodeSource( AgentToURL( agentClass, agentID ),
                                          new Certificate[0] );
  }
public static URL AgentToURL( String AgentClass, AgentID agentID )
  {return new URL( "http://" + agentID.place.domain +
                        "/" + agentID.place.place +
                        "/" + AgentClass +
}
```

Esempio di Secure Class Loader (2.)

```
protected Class findClass(final String className) throws ClassNotFoundException
{ …..
  return myFindClass( className );
}
private Class myFindClass( String className) throws Exception
{if
    la classe si trova in locale then
          byte [] classData= ...loadClassFile();
          verifico l'autenticità delle firme sul codice
          defineClass( className, classData, 0, classData.length, codeSource );
 else
    la cerco in remoto
    …….
}
```

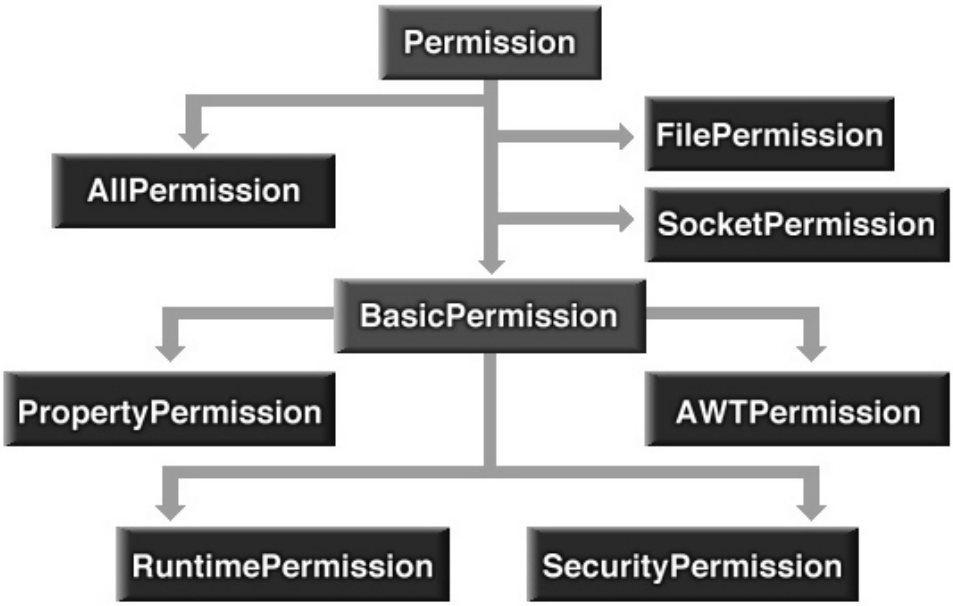

Permission Class

- I permessi sono caratterizzati da:
  - un type: che tipo di permessi (permessi relativi a file, a socket..);
  - un nome: identifica l'oggetto a cui i permessi sono riferiti
  - azioni
- 11 permessi standard Java, ognuno dei quali implementato come classe

Esempio: il FilePermission class

```
FilePermission p1 = new FilePermission ("-", "execute");
FilePermission p2 = new FilePermission ("/myclasses/*", "read");
```

Struttura della gerarchia



Controllo dell'Accesso

JDK 1.1

```
SecurityManager security=System.getSecurityManager();
if (security !=null) {
        Security.checkRead("path/file");
```

JDK 1.2

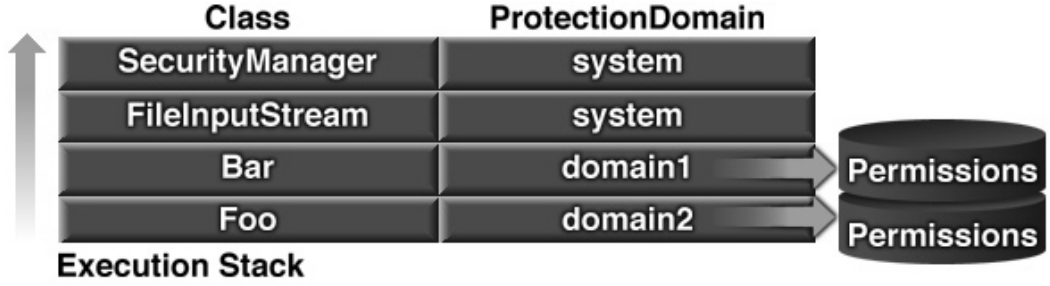
```
FilePermission perm = new
FilePermission("path/file","read");
AccessController.checkPermission (perm);
```

Algoritmo di Controllo dell'Accesso (1.)

- Algoritmo di controllo dell'accesso basato su stack inspection
- L'AccessController controlla se tutti i domini nella catena delle chiamate hanno il permesso richiesto
- strategie di implementazione: **lazy evaluation**

Algoritmo di Controllo dell'Accesso (2.)

- quando nello stack sono presenti più domini, tutti devono avere il permesso richiesto



Caratteristiche del modello di sicurezza nel JDK 1.3

Vantaggi:

- separazione tra specifica delle politiche ed effettivo enforcement
- estensibilità e scalabilità dei controlli di sicurezza
  (checkRead vs. checkPermission) => controllo dell'accesso "tipizzato"

Limiti:

- aggiornamento delle politiche semi-statico
- difficoltà di controllo dell'uso delle risorse (CPU..) => necessità di sistema di monitoring e accounting

Caratteristiche del modello di sicurezza nel JDK 1.4

Vantaggi:

- separazione tra specifica delle politiche ed effettivo enforcement
- estensibilità e scalabilità dei controlli di sicurezza
  (checkRead vs. checkPermission) => controllo dell'accesso "tipizzato"
- **aggiornamento delle politiche dinamico**

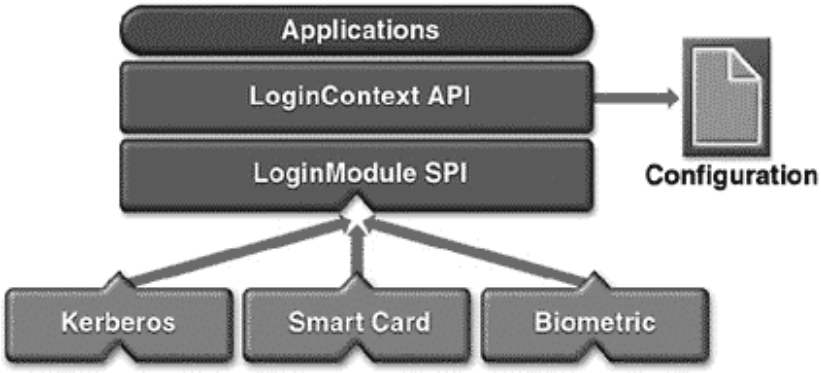
Limiti:

- difficoltà di controllo dell'uso delle risorse (CPU..) => necessità di sistema di monitoring e accounting

Java Authentication and Authorization Service (1.)

Obiettivi:
- autenticazione flessibile basata su plug-in dinamicamente installabili e configurabili
- autorizzazione basata sul code-source, sull'utente e sul ruolo

Attualmente disponibile nel J2SDK, v1.4.



Java Authentication Service (2.)

File di configurazione:

```
Login1 {
   sample.SampleLoginModule required;
   com.sun.security.auth.module.NTLoginModule sufficient;
   com.foo.SmartCard requisite debug=true;
   com.foo.Kerberos optional debug=true;
};
```

```
public final class LoginContext
public LoginContext(String name)
public void login() // two phase
process
public void logout()
// get the authenticated subject
public Subject getSubject()
```

```
public interface LoginModule
boolean login(); // first phase
boolean commit(); // second phase
boolean abort();
boolean logout();
```

Java Authorization Service (2.)

```
grant <signer(s) field>, <codeBase URL>
            <Principal field(s)> {
              permission perm_class_name "target_name", "action";
              ....
              permission perm_class_name "target_name", "action";
            };
```

Campo Principal:

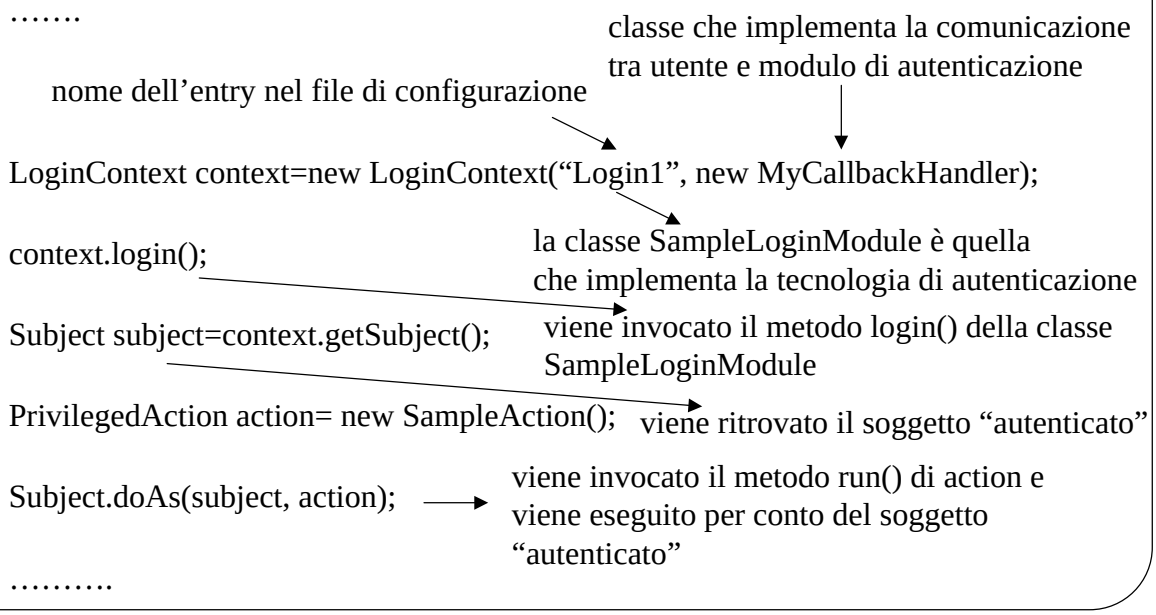
Principal Principal_class "principal_name"

Esempio:

```
grant codebase "file:./SampleAction.jar",
      Principal sample.principal.SamplePrincipal "testUser" {
              permission java.util.PropertyPermission "java.home", "read";
              permission java.util.PropertyPermission "user.home", "read";
              permission java.io.FilePermission "foo.txt", "read";
            };
```


Java Authentication and Authorization Service (3.)

Fasi necessarie per l'autenticazione ed autorizzazione:



Architettura Crittografica di Java (JCA) (1.)

Due principi:

- indipendenza dall'implementazione e interoperabilità
- estendibilità e indipendenza degli algoritmi

Due tipi di classi:

- una engine class è una classe astratta che dichiara le funzionalità di un dato tipo di algoritmo crittografico, senza però fornire alcuna implementazione.
- un provider è un package che implementa un certo insieme di funzionalità crittografiche.

Architettura Crittografica di Java (JCA) (2.)

Caratteristiche:

• più provider, anche di diversi produttori, possono coesistere ed interoperare

• il provider di default si chiama sun, ed fa parte del JDK

• una installazione di Java (JDK o JRE) può avere più provider installati: altri possono essere aggiunti dinamicamente, tramite il metodo Security.addProvider()

• un programma può richiedere genericamente una implementazione di un dato algoritmo (senza curarsi di quale provider la fornisca) o, invece, specificare il provider desiderato.

Esempio:

MessageDigest, Signature e KeyPairGenerator sono tre diverse engine class, che definiscono rispettivamente le funzionalità di digest, signature e di generatore di coppie di chiavi, secondo uno degli algoritmi a chiave pubblica disponibili

Struttura "a livelli" della JCA

La struttura della JCA è a due livelli:

- la JCA "di base" che definisce la cornice generale per la crittografia, definendo le classi astratte ma fornendo solo alcune semplici implementazioni di funzionalità specifiche
- la Java Cryptography Extension (JCE) che definisce le API complete e fornisce l'implementazione delle funzionalità di cifratura e decifratura.

ATTENZIONE:

a causa delle limitazioni all'esportazione di tecnologia crittografica fuori dagli Stati Uniti, la JCE della Sun non è esportabile.

- Senza limiti di export: http://jcewww.iaik.tu-graz.ac.at/

Bibliografia

- Li Gong, Inside Java 2 Platform Security, Addison Wesley
- Gary McGraw, E. Felten, Securing Java, Wiley
- articoli in:

  http://www.lia.deis.unibo.it/Courses/RetiDiCalcolatori/articolisicurezza.zip