ASPETTI DI SICUREZZA IN JAVA

- **Linguaggio privo di costrutti "unsafe"**
  - **distinzione fra applicazioni ed applet**
  - **linguaggio "strongly typed" e "type safe"**
  - **niente distruzioni esplicite di oggetti** (c'è il garbage collector), **niente accessi ad array senza controlli sugli indici,** etc.
- **Meccanismi a livello di supporto a run-time**
  - **macchina virtuale con verifica di bytecode**
  - ***class loader* e *security manager***
- **Modelli di sicurezza per applicazioni e applet**
  - **il modello base di Java 1.x *("sandbox")***
  - **il modello evoluto di Java 2**
- **Architettura crittografica (JCA + JCE)**

COSTRUTTI A LIVELLO DI LINGUAGGIO e MECCANISMI A LIVELLO DI SUPPORTO A RUN-TIME

COSTRUTTI A LIVELLO DI LINGUAGGIO

- **Ogni entità del linguaggio ha un *livello di protezione*** (private, default, protected, public), che può essere verificato
- **Assenza completa di puntatori e impossibilità di forgiarli**
- **Impedimento all'uso di variabili *non inizializzate***
- **Verifica che le funzioni siano sempre *chiamate* con il giusto numero e tipo di argomenti**
- **Verifica che le funzioni *restituiscano sempre un risultato* e che esso sia *coerente* con il tipo di ritorno dichiarato**
- **Impedimento di *cast arbitrario* tra tipi (primitivi e non)**
- **Verifica che ogni oggetto sia sempre usato per ciò che è e non si tenti mai di farne un uso diverso.**

VERIFICA DEL BYTECODE A LIVELLO DI CARICAMENTO

Perché un controllo a livello di caricamento?

- **un compilatore ben fatto può certamente controllare il rispetto delle regole precedenti…**
- **ma che succede se si costruisce un compilatore *malevolo* che *non lo fa?***

VERIFICA DEL BYTECODE CARICATO:

- **ri-verifica il rispetto della regole precedenti, *prima dell'esecuzione***
- **maggiore sicurezza, migliori prestazioni** (si evitano controlli durante la fase di esecuzione vera e propria)

VERIFICHE A LIVELLO A RUN-TIME

- **Controllo di accesso alle *variabili* in memoria, nel rispetto del livello di protezione di ognuna**
- **Controllo di accesso alle *locazioni* di memoria**
- **Controllo di assenza di *stack overflow / underflow***

ATTENZIONE però alla SERIALIZZAZIONE

- **un'entità salvata su disco è di fatto *accessibile,*** perché non protetta dal sistema di run-time di Java
- **CURA: introduzione del qualificatore `transient`**
  Ciò che è dichiarato transiente *non viene mai serializzato* e quindi ***non esce dall'ambiente protetto di run-time***

ESEMPIO DI VERIFICA

Classe corretta (frammento):

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

Classe "falsificata" (frammento):

```
public class MyCreditCard{
  public String acctNo;
  ...
}
```

IL FALSO VIENE SCOPERTO:

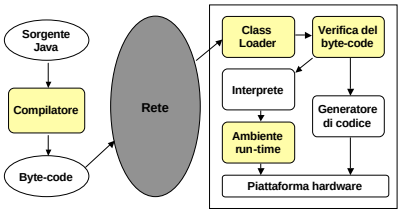
questo codice dà errore di compilazione.

```
CreditCard cc = Wallet.getCreditCard();
MyCreditCard carta = (MyCreditCard) cc;
System.out.println("N° carta: " + carta.acctNo);
```

CONTROLLI DI "TYPE SAFETY"

QUANDO vengono svolti i controlli?

1. **a tempo di compilazione**
2. **a tempo di caricamento delle classi**
3. **a tempo di esecuzione.**



1. CONTROLLI A TEMPO DI COMPILAZIONE

Il compilatore controlla che:

- **si acceda alle variabili solo quando permesso;**
- **non si acceda alla memoria in modo arbitrario;**
- **non si usino variabili non inizializzate**
- **la "maggior parte" dei cast sia corretta.**

Esempio di cast verificato a tempo di compilazione:

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

Alcuni cast però non sono controllati, in quanto:

- **non sempre si hanno tutte le informazioni necessarie (caricamento "lazy" delle classi, vedi oltre)**
- **si tende a posporre lavoro "probabilmente" inutile ad esempio, le parti di codice nei blocchi "catch"**

2. CONTROLLI A TEMPO DI CARICAMENTO

Ogni classe viene sottoposta a *bytecode verification*

- si controlla che una certa sequenza di bytecode rappresenti un insieme legale di istruzioni Java
  - **il file `.class` ha un formato corretto?**
  - **ogni classe eredita da una superclasse, come previsto?**
  - **esistono conversioni illegali tra tipi primitivi ?**

Se ne occupa il *Bytecode Verifier*

- una parte interna della Java Virtual Machine, **priva di interfaccia** – nessuno può interagire con essa né accedere ad alcuna delle sue parti
  - **alcune parti del bytecode, però, non vengono verificate, per motivi di efficienza (esempio: entro i blocchi *catch*)**

3. CONTROLLI A TEMPO DI ESECUZIONE

A run-time, si controllano:

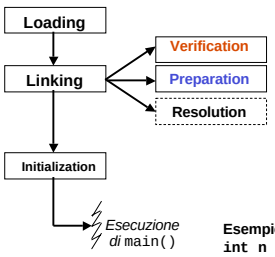
- **il rispetto degli estremi degli array**
- **gli altri cast tra oggetti.**

ESEMPIO

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

Se fallisce → `ArrayIndexOutOfBoundsException`

JAVA VIRTUAL MACHINE: FASI



caricamento in memoria del file *.class*

verifica byte-code

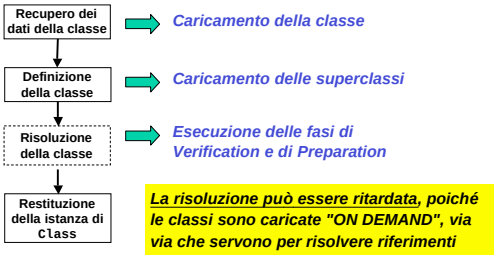
allocazione spazio in memoria

risoluzione di tutti i riferimenti simbolici ad altre classi

inizializzazione di tutte le variabili statiche allocate

Esempio:
`int n = 5`
preparation: allocazione memoria per l'`int`
inizialization: il valore di **n** viene posto a 5

CLASS LOADING



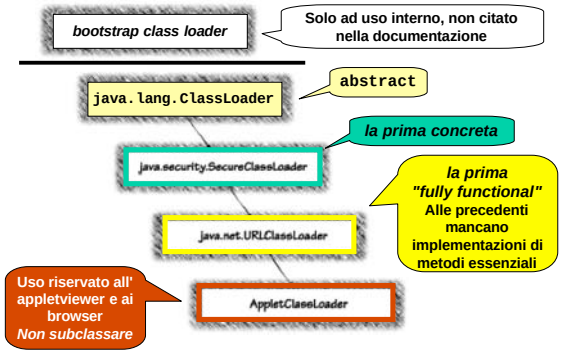
La risoluzione può essere ritardata, poiché le classi sono caricate "ON DEMAND", via via che servono per risolvere riferimenti (LAZY LOADING)

TASSONOMIA DI CLASS LOADER

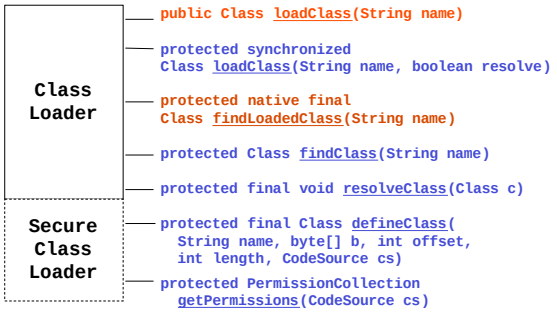
- ***Bootstrap class loader:*** **carica *le classi di sistema* *(core classes)* di Java, inclusi gli altri Class Loader**
- ***Class Loader:*** **classe astratta, radice della gerarchia**
- ***Secure Class Loader:*** **è la prima classe concreta, che applica le politiche di sicurezza.**
- ***URL Class Loader:*** **la vera classe "fully functional", che carica le classi applicative (appoggiandosi alla prec.)**
- ***Applet Class Loader:*** **classe riservata Sun per il caricamento delle applet nell'appletviewer e nei browser. *Non va usata per derivare sottoclassi*.**

È possibile definire propri tipi di class loader, estendendo ClassLoader, SecureClassLoader o URLClassLoader

TIPI DI CLASS LOADER: TASSONOMIA



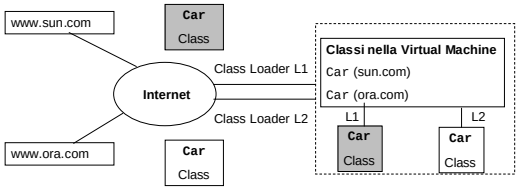
ClassLoader e SecureClassLoader



CLASS LOADER E SICUREZZA

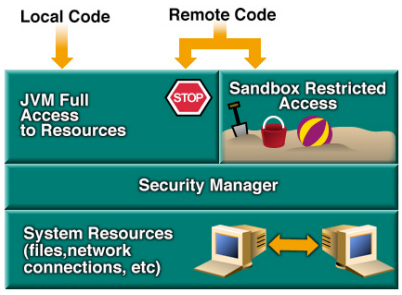
Il Class Loader gioca un ruolo fondamentale:

- **crea spazi di nomi separati**
- **si coordina con il *Security Manager* per implementare le politiche di sicurezza.**

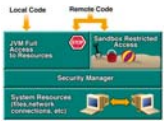


MODELLI DI SICUREZZA PER APPLICAZIONI E APPLET: da Java 1.0 a Java 1.x

IL PRIMISSIMO MODELLO DI SICUREZZA (Java 1.0)



IL PRIMISSIMO MODELLO DI SICUREZZA (Java 1.0)



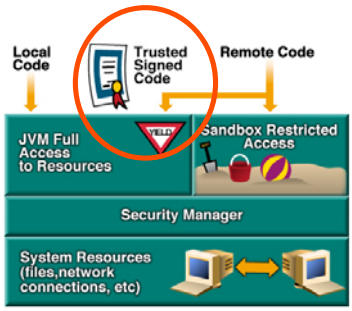
Vantaggi

- 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.

Limiti

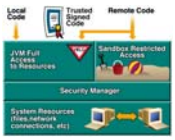
- modello troppo restrittivo
- i **programmatori di applicazioni** (non di applet) **DEVONO** scrivere un proprio Security Manager per aprire la sandbox
- ad ogni politica corrisponde una nuova versione del SecurityManager → proliferazione dei SecurityManager

IL MODELLO DI SICUREZZA DI Java 1.x



IL MODELLO DI SICUREZZA DI Java 1.x

DIFFERENZA rispetto al precedente:



- **il codice remoto può avere accesso pieno alle risorse purché sia *firmato***
- **la firma digitale del codice assicura *autenticazione* (dell'origine) e *integrità* (del codice eseguito)**

Limiti

- le applicazioni locali non sono sottoposte ad alcun controllo
- **tutte le classi presenti nel CLASSPATH si assumono *fidate:* non si distingue fra *classi realmente "di sistema"* e *classi applicative installate nel classpath***

SECURITY MANAGER

Una applicazione Java per default NON HA alcun Security Manager attivo, quindi *tutto è permesso.*

Al contrario, una applet Java è sottoposta ai vincoli dell' *AppletSecurityManager* e quindi:

- **non può accedere al file system**
- **non può accedere alla rete** (salvo l'URL da cui l'applet stessa è stata scaricata)
- **non può leggere molte proprietà di sistema**
- **non può eseguire altri programmi**
- **…**

RIDEFINIZIONE di SECURITY MANAGER

In Java 1.x, per *personalizzare le politiche* occorre definire e installare un proprio `SecurityManager` personalizzato.

Per fare ciò, occorre *definire un'opportuna sottoclasse* di `SecurityManager` e ridefinire gli opportuni metodi.

Attenzione però: i metodi di default di `SecurityManager` *lanciano sempre eccezione,* quindi occorre *ridefinirli tutti* (magari con un corpo vuoto).

Per questo, si adotta spesso un **approccio in due passi**:

- una classe `NullSecurityManager` che ridefinisce tutti i metodi attribuendo loro un **corpo vuoto** (non lancia più eccezioni)
- il proprio security manager derivato dalla precedente, così da **ridefinire *solo i metodi che realmente interessano.***

ESEMPIO (1/3): un security manager che blocca le verifiche sui file diversi da .tmp

```
public class MySecurityManager extends NullSecurityManager {
  final static String filenameEnding = ".tmp";
  public void checkRead(String file) {
    if (!file.endsWith(filenameEnding)) {
      throw new SecurityException("Read attempt: " + file);
    }
  }
  public void checkRead(String file, Object context) {
    checkRead(file);
  }
  public void checkWrite(String file) {
    if (!file.endsWith(filenameEnding)) {
      throw new SecurityException ("Write attempt: " + file);
    }
  }
}
```

ESEMPIO (2/3): un test di prova

```
public static void main (String args[]) {
   try {
     File f = new File("joe.txt");
     System.out.println ("joe.txt exists? " + f.exists() );
   } catch (SecurityException e) {
     System.err.println ("Cannot check if joe.txt exists");
   }
   System.setSecurityManager( new MySecurityManager() );
   System.out.println("Nuovo security manager installato");
   try {
     File f = new File("joe.txt");
     System.out.println ("joe.txt exists? " + f.exists() );
   } catch (SecurityException e) {
     System.err.println ("Cannot check if joe.txt exists");
   }
...
```

questo controllo si può ancora fare

questo non più!

scatta l'eccezione

ESEMPIO (3/3): il risultato del test

```
---------- Java Run ----------
joe.txt exists? true
After new SM Installed
Cannot check if joe.txt exists

---------- Java Run ----------
joe.tmp exists? true
After new SM Installed
joe.tmp exists? true
```

qui il controllo si poteva ancora fare

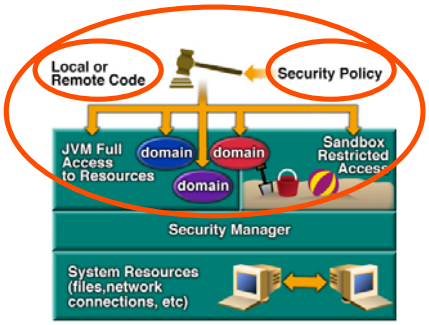
Dopo l'installazione del security manager, non più!

qui il controllo si poteva ancora fare

Si può fare anche dopo l'installazione del security manager, perché esso premette le verifiche sui file TMP

MODELLI DI SICUREZZA PER APPLICAZIONI E APPLET: il modello della piattaforma JAVA 2

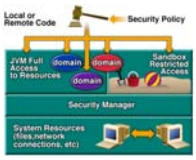
IL MODELLO DI SICUREZZA DI JAVA2



IL MODELLO DI SICUREZZA DI JAVA 2

OBIETTIVI

- Controlli di sicurezza su applet e *applicazioni, locali* e *remote*
- *Politiche di sicurezza configurabili* in modo *semplice*
- *Controlli di accesso* a granularità *fine*



CONCETTI BASE DELLA NUOVA ARCHITETTURA

- Definizione di permessi di accesso
- Politica di sicurezza e dominio di protezione
- Non occorre più *cambiare il security manager*, perché il suo comportamento può essere "guidato" da fuori

SICUREZZA IN JAVA2: CARATTERISTICHE

Vantaggi

- Separazione tra specifica delle politiche ed effettivo enforcement delle stesse
- Estendibilità e scalabilità dei controlli di sicurezza
- Controllo di accesso "tipizzato"

Limiti (e loro evoluzione)

- (v. < 1.4) valutazione delle politiche semi-statica
  (v. ≥ 1.4) valutazione delle politiche *lazy*
- Alcune difficoltà nel controllo d'uso delle risorse (cpu, …)

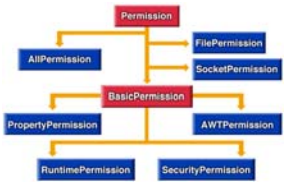
DOMINIO DI PROTEZIONE

Un *dominio di protezione* descrive gli oggetti accessibili da una data entità computazionale e con quali permessi.
Ogni classe appartiene a un dominio: il runtime mantiene il mapping fra codice della classe e dominio associato

Perciò, un *dominio di protezione* incapsula:

- **un *CodeSource* che descrive l'origine del codice e gli eventuali certificati digitali che ne garantiscono autenticità, etc.**
- **l'elenco delle entità computazionali che eseguono quel codice in un determinato istante**
- **un riferimento al ClassLoader che ha definito la classe**
- **l'elenco dei permessi garantiti staticamente all'atto del caricamento della classe; i permessi dinamici per un dominio sono determinati consultando la corrispondente politica.**

PERMESSI: GERARCHIA



`java.io.FilePermission`
per l'accesso al file system
`java.net.SocketPermission`
per l'accesso alla rete
`java.lang.PropertyPermission`
per le proprietà (properties) di Java
`java.lang.RuntimePermission`
per l'accesso a risorse di runtime
`java.security.NetPermission`
per l'autenticazione
`java.awt.AWTPermission`
per l'accesso alle risorse grafiche

Esempi di `FilePermission`:

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

SECURITY MANAGER e ACCESS CONTROLLER

`AccessController` costituisce una implementazione di default di `SecurityManager` che implementa tutto il nuovo schema di permessi e politiche.

In Java2, il metodo `checkPermission` di `SecurityManager` delega infatti quello omonimo di `AccessController`.

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

In Java2, perciò, **NON si realizzano più i propri Security Manager per sostituirli a quello dato: se ne personalizza il comportamento specificando apposite *POLITICHE DI SICUREZZA* definite da idonei *INSIEMI DI PERMESSI.***

IL MODELLO DI SICUREZZA IN JAVA 2: POLITICHE DI SICUREZZA, POLICY FILE E LO STRUMENTO *POLICYTOOL*

POLITICA DI SICUREZZA

Una *politica di sicurezza* è una matrice di controllo di accesso: specifica il tipo di accesso consentito a chi e in quale situazione, con la granularità desiderata.
ESEMPIO:

codice	permessi	utente
applet1	read, write `/home/enrico`	
altre applet firmate	read, write `/temp`	
tutti	read, write `/`	enrico

La politica di sicurezza è descritta da un file di testo detto *policy file,* in un formato standardizzato.
Internamente al sistema, la politica di sicurezza in uso è rappresentata da un oggetto di classe **`Policy`**.

IL POLICY FILE

Il policy file specifica quali permessi garantire a un certo codice (espresso come URL) ed eventualmente anche se dev'essere firmato digitalmente e *da chi*.

ESEMPIO BASE (senza firme digitali)

```
grant
  CodeBase "http://mioindirizzo/miopath/" {
  permission java.io.FilePermission "read, write", "tmp/*";
};
```

- Questo policy file garantisce al codice scaricato dall'URL indicato il permesso di lettura e scrittura *nella directory tmp*
- **E' possibile indicare più righe `permission` nell'ambito della medesima specifica `grant`.**

POLICY FILE DI SISTEMA E DI UTENTE

La politica di sicurezza è l'unione di due specifiche:

- il policy file di sistema, che si chiama `java.policy` e si trova nella directory `C:/j2sdk1.5/jre/lib/security`

+

- l'eventuale policy file personale dell'utente, di nome `.java.policy`, posto nella home directory dell'utente.

NOTA

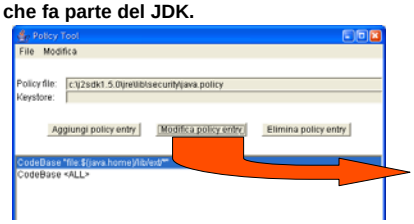
Il nome e la posizione del policy file di sistema *non sono imposti:* il nome e la posizione del policy file di sistema sono infatti impostate dal *security file* (`java.security`), che contiene una riga della forma

`policy.java=java.policy`

dove il nome a destra dell'uguale è personalizzabile.

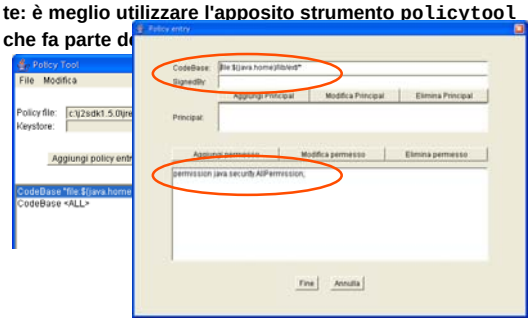
EDITING DEL POLICY FILE (1)

Il policy file non dovrebbe essere modificato direttamente: è meglio utilizzare l'apposito strumento `policytool` che fa parte del JDK.



EDITING DEL POLICY FILE (2)

Il policy file non dovrebbe essere modificato direttamente: è meglio utilizzare l'apposito strumento `policytool` che fa parte d



SINTASSI DEL POLICY FILE

- Ogni blocco `grant` garantisce un permesso al codice indicato da `CodeBase`
- Se si specifica in `signedBy` un elenco di firmatari, il permesso sarà concesso solo se tutti i firmatari avranno firmato il codice (non uno solo fra essi)
  Per indicare che basta un firmatario solo, occorrono tanti *blocchi `grant` separati* – uno per firmatario.

ESEMPIO

```
grant signedBy "enrico", "andrea"
  CodeBase "http://mioindirizzo/miopath/" {
  permission java.io.FilePermission "read, write", "tmp/*";
};
```

POLICY FILE e FIRME DIGITALI

- Gli alias dei firmatari usati nel policy file devono essere definiti in modo sicuro: per questo, a ciascun alias viene associata *la chiave pubblica* corrispondente.
- Tale associazione è mantenuta nel *keystore* dell'utente, ossia nel file cifrato `.keystore` posto nella home directory dell'utente (esiste anche un *keystore globale*, chiamato `cacerts`: non ce ne occupiamo)
  - l'accesso a tale file è protetto da password
  - opzionalmente, anche l'accesso a ogni singolo alias in esso contenuto può essere protetto da una password specifica
- A questo file si accede solo indirettamente, tramite lo strumento `keytool`.

ALTRI STRUMENTI PER LA SICUREZZA IN JAVA 2: LO STRUMENTO *KEYTOOL*

KEYTOOL

Lo strumento ***keytool*** permette di

- generare coppie di chiavi pubblica/privata (DSA, RSA) memorizzandole nel *keystore*: alla chiave pubblica è altresì associato un certificato X.509
  - auto-firmato (per uso privato)
  - generazione di una richiesta per la CA
    (si dovrà poi importare il certificato restituito firmato)
- memorizzare nel *keystore* certificati altrui, associando ad essi un alias.

NOTA
L'algoritmo predefinito è DSA (unico disponibile fino al JDK 1.3)
L'opzione `-keyalg` specifica altri algoritmi (es. `rsa`, da JDK 1.4)

KEYTOOL: GENERAZIONE DI CHIAVI

```
E:esercizi>keytool -keyalg rsa -genkey
Immettere la password del keystore:  ●●●●●●
Specificare nome e cognome
 [Unknown]:  Enrico Denti
Specificare il nome dell'unità aziendale
 [Unknown]:  DEIS
Specificare il nome dell'azienda
 [Unknown]:  Universita di Bologna
Specificare la località
 [Unknown]:  Bologna
Specificare la provincia
 [Unknown]:  BO
Specificare il codice a due lettere del paese
 [Unknown]:  IT
Il dato CN=Enrico Denti, OU=DEIS, O=Universita di Bologna,
L=Bologna, ST=BO, C=IT è corretto?
 [no]:  si
Immettere la password della chiave per <mykey>
(INVIO se corrisponde alla password del keystore):
```

alias di default

KEYTOOL: CONTROLLO

Lo strumento ha in effetti generato il file `.keystore` nella home directory dell'utente:

```
>dir "C:\Documents and Settings\Enrico Denti\.keystore"
Directory of C:\Documents and Settings\Enrico Denti
15/05/2004  11.23             1.370 .keystore
```

Controllo del contenuto del keystore:

```
E:esercizi>keytool -list
Immettere la password del keystore: ●●●●●●
Tipo keystore: jks
Provider keystore: SUN
Il keystore contiene 1 entry
mykey, 15-mag-2004, keyEntry,
Impronta digitale certificato (MD5):
63:A4:9E:3D:A2:99:9D:EE:54:E9:A2:F1:43:84:38:66
```

KEYTOOL: generazione ed export di certificati

Per esportare il proprio certificato su file:

```
E:esercizi>keytool -export -file mycert
Immettere la password del keystore: ●●●●●●
Il certificato è memorizzato nel file <mycert>
```

Per ristamparlo come verifica:

```
E:esercizi>keytool -printcert -file mycert
Immettere la password del keystore: ●●●●●●
Proprietario: CN=Enrico Denti, OU=DEIS, O=Universita di
Bologna, L=Bologna, ST=BO, C=IT
Organismo di emissione: CN=Enrico Denti, OU=DEIS,
O=Universita di Bologna, L=Bologna, ST=BO, C=IT
Numero di serie: 40a5e1a4
Valido da Sat May 15 11:23:48 CEST 2004 a Fri Aug 13 11:23:48
CEST 2004
Impronte digitali certificato:
MD5: 63:A4:9E:3D:A2:99:9D:EE:54:E9:A2:F1:43:84:38:66
SHA1:67:AE:0A:05:78:25:50:F8:5E:17:EF:EE:12:F1:6F:22:B0:E4:0C:A7
```

KEYTOOL: generazione ed export di certificati

Per esportare il proprio certificato su file:

```
E:esercizi>keytool -export -file mycert
Immettere la password del keystore: ●●●●●●
Il certificato è memorizzato nel file <mycert>
```

Per ristamparlo come verifica:

```
E:esercizi>keytool -printcert -file mycert
Immettere la password del keystore: ●●●●●●
Proprietario: CN=Enrico Denti, OU=DEIS, O=Universita di
Bologna, L=Bologna, ST=BO, C=IT
Organismo di emissione: CN=Enrico Denti, OU=DEIS,
O=Universita di Bologna, L=Bologna, ST=BO, C=IT
Numero di serie: 40a5e1a4
Valido da Sat May 15 11:23:48 CEST 2004 a Fri Aug 13 11:23:48
CEST 2004
Impronte digitali certificato:
MD5: 63:A4:9E:3D:A2:99:9D:EE:54:E9:A2:F1:43:84:38:66
SHA1:67:AE:0A:05:78:25:50:F8:5E:17:EF:EE:12:F1:6F:22:B0:E4:0C:A7
```

Questo certificato è in formato standard (ASCII a 7 bit)
Può essere inviato ad altri via mail
Può essere posto sulla propria pagina web per consentire a chiunque di verificare le proprie credenziali

KEYTOOL: import di certificati

L'alias di default usato da keytool per memorizzare nel keystore il proprio certificato è ***mykey***

Quando si importa un certificato (proprio o altrui) occorre quindi specificare un alias
Tale alias viene usato, tra gli altri, dal policy file.

Per importare un certificato da un file, associandolo all'alias `pippo`:

```
E:esercizi>keytool -import -alias pippo -file mycert
Immettere la password del keystore: ●●●●●●
Il certificato è stato aggiunto al keystore
```

LA SITUAZIONE ATTUALE IN ITALIA: CERTIFICATI e AUTORITÀ DI CERTIFICAZIONE

CERTIFICATI DI PROVA

Alcune Certification Authority offrono l'emissione di *certificati di prova gratuiti* (tipicamente da 30-60 giorni)

- **Trust Italia (Verisign):** `www.trustitalia.it` *(60 gg)*
- **E-Trustcom** `etrust.intesa.it` *(30 gg)*

Essi costituiscono un valido mezzo per

- sperimentare un servizio
- verificare il buon funzionamento di un'applicazione

ALCUNI CERTIFICATORI anche per singoli

Anche solo emissione certificati per browser/email:

- **Trust Italia (Verisign):** `www.trustitalia.it`
- **E-Trustcom** `e-trustcom.intesa.it`

Solo kit completi (con riserva di verifica!!):

- **Infocamere:** `www.card.infocamere.it`
- **I.T. Telecom:** `www.firmasicura.it`
- **Poste Italiane:** `www.poste.it/online/postecert`
- **Actalis:** `www.actalis.it`

CLASSI DI CERTIFICATI

CLASSE 1 (minimo)

1. **non ha validità legale, ok fra singoli (entro certi limiti)**
2. **la CA verifica solo che l'indirizzo email associato al certificato esista e sia sotto il controllo dell'intestatario, ma *non garantisce l'identità* della persona**

CLASSE 2

1. **ha validità legale**
2. **la CA assicura la corrispondenza fra la coppia di chiavi e la data persona, ma la modalità di emissione del certificato *non arriva a pretendere la presenza fisica* del richiedente**

CLASSE 3 (massimo)

1. **ha la massima validità legale**
2. **la CA pretende un documento di identità valido per la verifica dei dati forniti ed esige la *presenza fisica* del cliente presso una delle sedi della RA (o un documento autenticato da un notaio che provi l'identità del richiedente)**

ESEMPI DI CERTIFICATI PER SINGOLI (1)

Trust Italia (Verisign): `www.trustitalia.it`

- **Certificato annuale per singoli, classe 1: € 15.83 + IVA**
- **Certificato annuale per singoli, classe 3: € 42.35 + IVA**

Certificati legati in modo indissolubile all'indirizzo email, per:

- **Firma digitale di messaggi email e di documenti**
- **Invio e ricezione di messaggi cifrati**
- **Autenticazione dell'identità e dei privilegi dell'intestatario**

Livello di Autenticazione di CLASSE 1 (minimo)

- **ok per cifratura di email e autenticazione dell'intestatario.**
- **non indicato per usi commerciali, in tutti quei casi in cui sia richiesta la prova dell'identità degli operatori**

Livello di Autenticazione di CLASSE 3 (massimo)

- **adatto anche per operazioni e transazioni 'ad alto rischio' effettuate on-line, nelle applicazioni di commercio elettronico, trading on-line, home banking**

ESEMPI DI CERTIFICATI PER SINGOLI (2)

E-Trustcom `e-trustcom.intesa.it`

- **Costi dei certificati non specificati sul Web (per ora)**

La CA offre

Livello di Autenticazione di CLASSE 1 (minimo)

- **gratis, servizio offerto a solo scopo di demo**

Livello di Autenticazione di CLASSE 2

- **per aziende o enti che necessitano di abilitare una lista controllata di assegnatari a servizi quali la firma digitale di documenti**
- **durata massima: 1 anno**

Livello di Autenticazione di CLASSE 3 (massimo)

- **per aziende o enti che necessitano di abilitare una lista controllata di assegnatari sicura e controllata a servizi quali la firma digitale di documenti**

IL MODELLO DI SICUREZZA DI JAVA 2: APPLET vs APPLICAZIONI

APPLET e MODELLO DI SICUREZZA

Le applet non hanno i diritti di un'applicazione: la loro esecuzione è sorvegliata dall'AppletSecurityManager

- in generale, non possono accedere al file system,
- non possono accedere alla rete (salvo il sito da cui sono state scaricate),
- non possono leggere molto proprietà di sistema, ecc.

Quindi, una applicazione che sia anche applet e che, ad esempio, acceda al file system, avrà due comportamenti diversi secondo come viene invocata:

- se eseguita come applicazione, funzionerà
- se eseguita in un browser come applet non funzionerà, in quanto il security manager solleverà un'eccezione di sicurezza, impedendo l'inizializzazione.

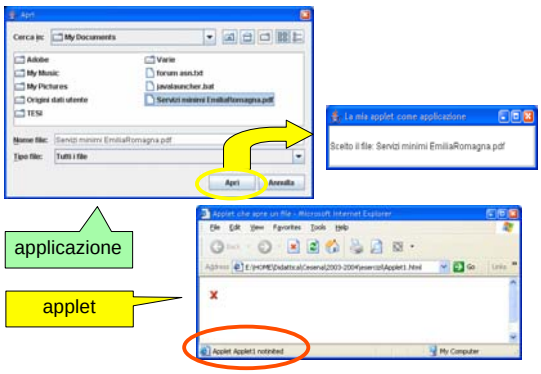
UNA APPLET DI ESEMPIO…

```
public class Applet1 extends JApplet {
 JTextField res;
 public void init(){
  Container c = getContentPane();
  risp = new JTextField(20); c.add(risposta);
  JFileChooser fc = new JFileChooser();
  int returnVal = chooser.showOpenDialog(null);
  if(returnVal == JFileChooser.APPROVE_OPTION)
    res.setText("Scelta: "+ fc.getSelectedFile().getName());
 }
 public static void main(String args[]){
  JApplet applet = new Applet1();
  JFrame f = new JFrame( "Applet come applicazione ");
  f.setSize(new Dimension(100,100));
  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  Container c = f.getContentPane();
  c.add(applet); applet.init(); applet.start();
  f.setVisible(true);
 }
}
```

…CON IL RELATIVO DOCUMENTO HTML

```
<HTML>
<HEAD>
<TITLE> Applet che apre un file </TITLE>
</HEAD>
<BODY>
  <APPLET CODE="Applet1.class" WIDTH=300 HEIGHT=100 >
</APPLET>
</BODY>
</HTML>
```

APPLET DI ESEMPIO: ESPERIMENTO



APPLET DI ESEMPIO: NUOVA POLITICA

Perché l'applet funzioni, occorre garantirle due permessi:

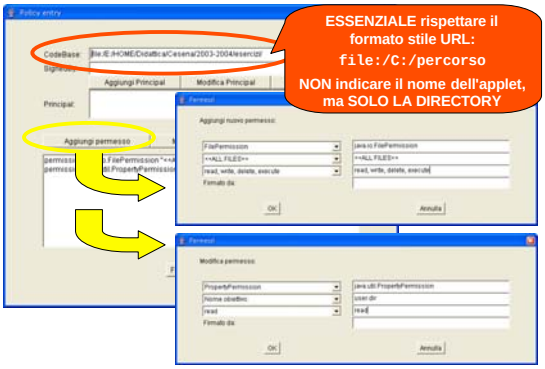
- `FilePermission` `read, write, execute`
- `PropertyPermission "user.dir"` `read`

La seconda è necessaria perché l'applet possa leggere il file `.java.policy` che stiamo per creare, che si troverà nella home directory dell'utente.

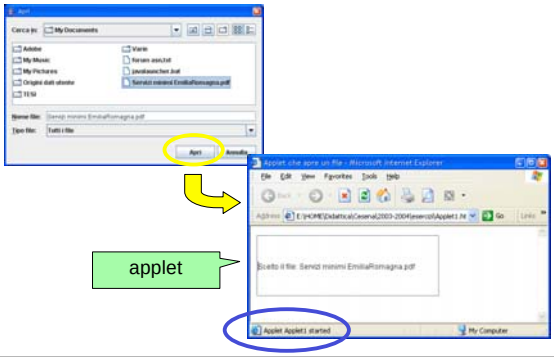
Per creare tale file, usiamo lo strumento *policytool* come segue, salvando il risultato nella directory opportuna: `C:\Documents and Settings\Enrico`

NB: questa directory può variare in base alla versione di Windows in uso: l'esempio è riferito a Windows XP.

APPLET DI ESEMPIO: NUOVA POLITICA



NUOVO ESPERIMENTO con il browser



ALTERNATIVA: esperimento con l'appletviewer

Lanciare un'applet con l'appletviewer può essere utile a *fini di debugging*, per vedere bene i messaggi d'errore.

Attenzione però: poiché di norma l'appletviewer NON APPLICA le politiche di sicurezza dei browser, occorre specificarle, altrimenti il test risulta falsato.

```
appletviewer -Jdjava.security.policy="C:\Documents
and Settings\Enrico\.java.policy" Applet1.html
```

- `-J` passa l'argomento seguente all'interprete Java
- `-D` imposta quella proprietà mentre lancia l'interprete Java

APPLET & PERMESSI: LIMITI

L'approccio precedente ha permesso di eseguire l'applet concedendole i permessi necessari.

PROBLEMI:

- **I permessi sono in realtà concessi a una directory (un URL)**
- **Chi mi garantisce che l'applet caricata sia autentica e non una malevola con lo stesso nome?**
  Come posso fidarmi a dare dei permessi a un'entità della cui origine e integrità non sono certo???

Per questi motivi, solitamente NON si danno permessi ad applet sulla base del solo URL:

si pretende che l'applet sia FIRMATA DIGITALMENTE.

IL MODELLO DI SICUREZZA DI JAVA 2: APPLET FIRMATE DIGITALMENTE

LO STRUMENTO *JARSIGNER*

APPLET FIRMATE

PASSI

- **la politica nel policy file deve esplicitare *DA CHI* dev'essere firmata l'applet**
- **l'applet deve necessariamente essere *inserita in un archivio JAR***
- **l'archivio JAR deve essere firmato digitalmente mediante l'apposito strumento `jarsigner`**

Quest'ultimo usa le chiavi memorizzate nel keystore, che si generano nel modo già visto in precedenza.

APPLET FIRMATE

Generazione del file JAR:

```
jar -cf Applet1.jar  Applet1.class
```

Firma del file JAR: l'alias del firmatario da usare

```
jarsigner Applet1.jar  mykey
Immettere la password del keystore: ●●●●●●
Warning: il certificato scadrà fra sei mesi
```

Eventuale verifica del file JAR (con o senza `-verbose`):

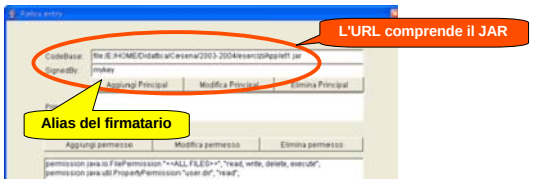
```
jarsigner -verify -verbose Applet1.jar
...
smk 1507 Mon May 17 12:35:44 CEST 2004 Applet1.class
  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
jar verified.
```

POLITICA CON APPLET FIRMATA

Modifiche nel policy file:

URL: `file:/...../esercizi/Applet1.jar`

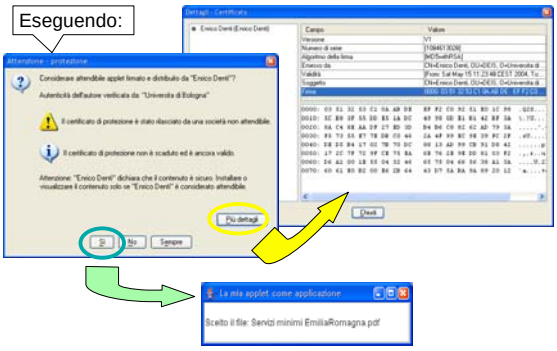
SIGNEDBY: `mykey`



IL DOCUMENTO HTML per l'APPLET FIRMATA

```
<HTML>
<HEAD>
<TITLE> Applet che apre un file </TITLE>
</HEAD>
<BODY>
<APPLET CODE="Applet1.class" ARCHIVE="Applet1.jar"
 WIDTH=300 HEIGHT=100 >
</APPLET>
</BODY>
</HTML>
```

APPLET FIRMATA: ESPERIMENTO



UN ULTIMO ASPETTO

L'approccio precedente ha permesso di firmare un JAR, in modo da garantire che il suo contenuto sia sicuro.

PROBLEMA

- **La firma garantisce il contenuto di quel JAR, ma *non impedisce che qualcuno definisca altre classi (non originali!) dichiarandole surrettiziamente parte dello stesso package.***

In altri termini, qualcuno potrebbe "estendere" il package aggiungendo classi sue (e quindi insicure) al JAR iniziale: ciò consentirebbe a tali classi di accedere a tutti i membri con visibilità di package, compromettendo la sicurezza.

Questo rischio si evita *SIGILLANDO il JAR:* in tal modo, *tutte le classi di quel package devono avere quell'unico JAR come origine.* Ogni altra origine causa eccezione.

IL MODELLO DI SICUREZZA DI JAVA 2: ARCHIVI JAR SIGILLATI DIGITALMENTE

JAR e PACKAGE SIGILLATI ("SEALED")

Si può scegliere di sigillare **tutto il JAR** (con ciò sigillando tutte le classi e i package ivi contenuti) **oppure solo uno o più** dei package in esso contenuti.

Per sigillare un package o un intero archivio JAR, occorre:

1. **predisporre un file MANIFEST che contenga la riga:**
   `Sealed: true` **o a livello generale** (per sigillare **tutto il JAR**) **o dopo aver specificato un nome di package**:
   `Name: nomepackage`
   `Sealed: true`
   In questo caso, si sigilla **solo quel package** entro il JAR.
2. **costruire l'archivio JAR fornendo quel MANIFEST**
3. **infine, firmare digitalmente il JAR come in precedenza.**

JAR SIGILLATI: ESEMPIO COMPLETO

Poiché il sigillo chiude un package, bisogna *averne uno.*

Ristrutturiamo perciò l'applicazione in questo modo:

- **il VERO package (`sealed`) contenente**
  - **l'applet (ridenominata `Applet1toseal`)**
  - **la classe con il main di test (`ProvaSealed`)**
- **un FINTO package omonimo (`sealed`), posto in un'altra directory (`badclass`) e contenente**
  - **la classe abusiva (`ExtraBadClass`)**

IL PACKAGE "ORIGINALE" (1/2)

```
package sealed;
public class Applet1toseal extends JApplet {
   ...
  public static void main(String args[]){
   JApplet applet = new Applet1toseal();
   ...
  }
}
```

Il main di prova:

```
package sealed;
public class ProvaSealed {
  public static void main(String[] args) {
   ...
  }
}
```

IL PACKAGE "ORIGINALE" (2/2)

Il main tenta di caricare ExtraBadClass dal finto package.
E' chiaramente una assurdità, ma "crea il caso" in modo evidente,

```
package sealed;
import java.net.*;
import java.security.*;
public class ProvaSealed {
  public static void main(String[] args) {
   try {
     URL[] url = new URL[]{new URL("file:badclass/")};
     URLClassLoader loader = URLClassLoader.newInstance(url);
     Class cl = loader.loadClass("sealed.ExtraBadClass");
     System.out.println(loader + " loaded " + cl);
     }
    catch (Throwable t) { t.printStackTrace(); }
  }
}
```

IL PACKAGE "FINTO"

Il finto package si chiama nello stesso modo, ma proviene da una diversa origine (diverso URL – qui, una diversa directory)

```
package sealed; // è quello finto!!
public class ExtraBadClass {
 // vuota
}
```

JAR SIGILLATI: CREAZIONE

Compilazione della classe "aliena":

```
E:esercizi\badclass>javac sealed/ExtraBadClass.java
```

Compilazione del package "originale":

```
E:esercizi>javac sealed/*.java
```

Contenuto del file MANIFEST preparato (`manifest.tmp`):

```
Sealed: true
```

Generazione del file JAR sigillato e firma dello stesso:

```
jar -cmf manifest.tmp Sealed1.jar  sealed/*.class
```

```
jarsigner Sealed1.jar  mykey
Immettere la password del keystore: ●●●●●●
Warning: il certificato scadrà fra sei mesi
```

JAR SIGILLATI: ESPERIMENTO

Ora eseguiamo la classe originale **`ProvaSealed`**, dando come percorso di ricerca dei file soltanto il JAR sigillato:

```
E:esercizi>java -cp Sealed1.jar sealed.ProvaSealed
java.lang.SecurityException: sealing violation:
package sealed is sealed
  at java.net.URLClassLoader.defineClass(Unknown Source)
  ...
```

Come si vede **scatta l'eccezione,** perché il class loader constata il tentativo da parte di una classe (qui, per comodità di test, è la classe che contiene il main a farlo) di caricarne una ***apparentemente* dello stesso package,**
***ma in realtà* proveniente da un altro URL.**

A METÀ FRA APPLET E APPLICAZIONI:
Java Net Launching Protocol (JNLP)
JAVA WEB START

FRA APPLET E APPLICAZIONI

Un'applet firmata è sicura e pratica, ma deve pur sempre essere eseguita *all'interno di un browser.*

Ciò comporta alcuni *limiti:*

- **carico aggiuntivo all'atto del caricamento**
- **limiti all'aspetto esteriore dell'interfaccia grafica, poiché l'applet è inglobata nella pagina web**
- **i comandi e i menù del browser si sovrappongono a quelli dell'applet → rischio di confondere l'utente**

La soluzione: *JNLP (Java Net Launching Protocol)*

un protocollo per scaricare ed installare una applicazione sulla macchina dell'utente
(o anche solo per eseguirla da remoto)

APPLICAZIONI JNLP

Una applicazione JNLP può essere:

- **scaricata e installata sul pc utente, *come un'applicazione***
  o anche
  eseguita da remoto, *come un'applet*
- **firmata digitalmente *come un'applet***
  o anche
  non firmata digitalmente, potendo comunque accedere a risorse locali dietro *consenso esplicito dell'utente* ma solo *usando apposite API* (non in modo diretto).

Una applicazione JNLP può essere lanciata in *tre modi:*
– **da *riga di comando,* con il comando `javaws`**
– **tramite una *icona sul desktop***
– **mediante l' *Application Manager JNLP.***

JNLP vs. JAVA WEB START

JNLP è un protocollo:
JAVA WEB START è la sua *implementazione ufficiale Sun.*

IMPORTANTE:

- **il Web server utilizzato deve supportare il nuovo tipo MIME `application/x-java-jnlp-file`, associandolo ai file di estensione `jnlp`**
- Tomcat è già configurato correttamente
- **Apache richiede l'aggiunta nel file `mime.types` situato nella directory `conf` della riga di configurazione:**

```
application/x-java-jnlp-file   jnlp
```

COSTRUIRE UN'APPLICAZIONE JNLP

Per costruire un'applicazione JNLP, occorre:
- **costruire l'applicazione come sempre**
- **archiviarla obbligatoriamente in un file JAR** (firmato o no)
- **scrivere un opportuno *file di avvio* in formato XML, che *spieghi al client come si scarica e installa l'applicazione.***

Una volta scritto il file di avvio, non rimane che porre in una pagina Web un normale link a tale file; ad esempio,

```
<a href="Prova.jnlp"> Avvia l'applicazione </a>
```

SUL SERVER WEB devono quindi esserci tre file:
- **la pagina web (ad esempio, `Prova.html`)**
- **l'applicazione (ad esempio, `Prova.jar`)**
- **il file di avvio (ad esempio, `Prova.jnlp`)**

APPLICAZIONI JNLP FIRMATE e NON FIRMATE

- Se il file JAR è firmato, l'applicazione può accedere alle risorse locali DIRETTAMENTE, senza vincoli
- Se il file JAR non è firmato, l'applicazione può accedere alle risorse locali SOLO IN MODO INDIRETTO, tramite le JNLP API che fungono da *mediatore* e nascondono la implementazione locale del servizio richiesto.

JNLP API

- Unica classe, `ServiceManager`, che definisce il metodo statico `lookup` per accedere ai servizi per nome
- Molte interfacce, tra cui: `DownloadService`, `FileContents`, `FileOpenService`, `FileSaveService`, `PrintService`, etc.

Dal JDK 1.5, Java Web Start è parte della distribuzione standard

UN ESEMPIO DI FILE DI AVVIO (1/2)

IPOTESI:
Applicazione JNLP archiviata nel file `Prova.jar`

Occorre scrivere un file XML di nome `Prova.jnlp`
da porre nella *stessa directory del file JAR sopra citato*
e il cui root tag sia `<jnlp>`

```
<?xml version="1.0" encoding="UTF-8" ?>
<jnlp codebase="http://..." href="Prova.jnlp">
  <information> ... ... </information>
  <resources> ... ... </resources>
  <security> ... ...  </security>
  <application-desc main-class="Prova" />
</jnlp>
```

URL del file

permessi (solo JAR firmati)

UN ESEMPIO DI FILE DI AVVIO (2/2)

```
<information>
  <title> ... </title>
  <vendor> ... </vendor>
  <description> ... </description>
  <icon-href="..."/>
  <offline-allowed/>
</information>
```

Campi usati dagli strumenti di amministrazione di Java Web Start per installare l'applicazione e i relativi collegamenti

```
<resources>
  <j2se version="1.4+"/>
  <jar  href="Prova.jar"
        download=""eager"/>
</resources>
```

EAGER: il JAR dev'essere tutto scaricato prima di iniziare l'esecuzione. LAZY: si può iniziare l'esecuzione anche prima.

```
<security>
  <all-permissions/>
</security>
```

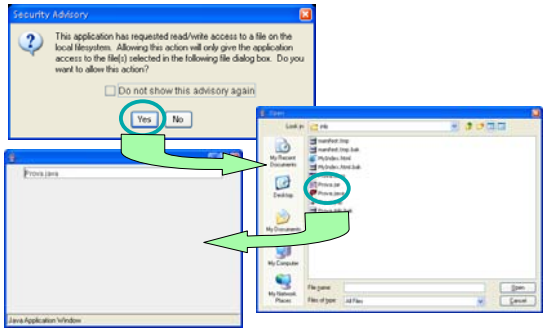
SOLO PER APPLICAZIONI FIRMATE: specifica i permessi di esecuzione

UNO SKETCH DI APPLICAZIONE

```
import javax.jnlp.*;
...
FileOpenService fs = null;
try {
  fs = (FileOpenService) ServiceManager.lookup(
                              "javax.jnlp.FileOpenService");
}
catch (UnavailableServiceException e1) { ... }
if (fs!=null) { // ok, abbiamo accesso al servizio
  try {
    FileContents c = fs.openFileDialog(".", new String[]{"*"});
    if (c==null) return;
    output.setText(c.getName());
  }
  catch (Exception e2) { ... }
}
```

TEST DA RIGA DI COMANDO con `javaws`

```
E:>javaws http://deis127.deis.unibo.it/enrico/Prova.jnlp
```



TEST via BROWSER (su server Apache)