Laurea Specialistica in Ingegneria Informatica
Università degli studi di Bologna
A.A. 2004/05

Java Security Extensions

Tecnologie per la Sicurezza LS
Prof. R. Laschi

Maurizio Colleluori
colleluori_mz@libero.it

Java Security Extensions

- Con la versione 1.4 della Java 2 Standard Edition sono state apportate modifiche significative alle iniziali caratteristiche di sicurezza del linguaggio
  - Sono stati integrati in un'unica Java 2 Security Platform il framework di base per la sicurezza (JCA) e tre Security Extensions in precedenza fornite come opzionali:

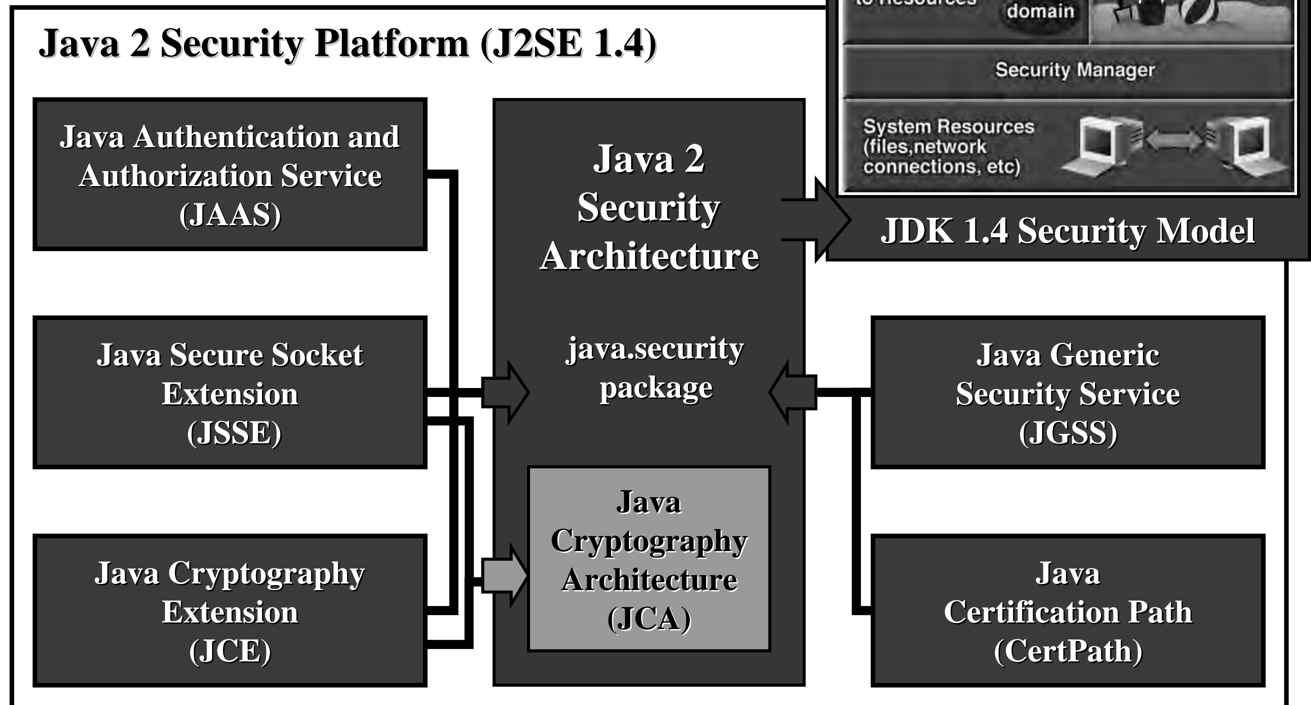
    - **JCE (Java Cryptography Extension)**
    - **JSSE (Java Secure Socket Extension)**
    - **JAAS (Java Authentication and Authorization Service)**

  - Sono stati inoltre inseriti ex novo i seguenti packages:

    - **Java CertPath API (Java Certification Path)**
    - **JGSS API (Java Generic Security Service)**

- Anche il JDK 1.4 Security Model prevede nuove importanti caratteristiche:
  - adattamento della gestione dei permessi al modello JAAS
  - gestione dinamica delle politiche di sicurezza (Dynamic Security Policy)
  - miglioramento della sicurezza nella gestione delle applet

Java Security Platform



JDK 1.4 Security Model

- Evoluzione rispetto al modello di sicurezza precedente (Sandbox):
  - non più modello ON/OFF (la classe può fare o tutto o nulla)
  - maggiore granularità (insieme di permessi)
  - maggiore flessibilità (permessi in base alle esigenze)
- Punti cardine:
  - provenienza del codice (host remoto o file system locale)
  - quali permessi, per quali operazioni e per quali utenti (*"chi può fare che cosa e in quali circostanze"*)
  - aggiornamento dinamico delle politiche di sicurezza (Dynamic Security Policy)
- Componenti:

Access Controller	Access Permissions
Code Source	Protection Domain
Security Manager	Security Policy

JCA
Java Cryptography Architecture

Caratteristiche

- La Java Cryptography Architecture (JCA) rappresenta il framework di base per la crittografia ed appartiene all'ambiente run-time
- E' costruita intorno alla Java 2 Security Architecture
- Si basa sui principi della Cryptographic Service Provider Architecture:
  - indipendenza (dal tipo di implementazione)
  - interoperabilità
  - estendibilità
- A tal fine impiega due tipologie di classi:
  - classi astratte di tipo ***engine*** che dichiarano le funzionalità di un dato tipo di algoritmo crittografico
  - classi di tipo ***provider*** che implementano un certo insieme di funzionalità crittografiche per un Cryptographic Service Provider (CSP)
- Caratteristiche della CSP Architecture:
  - una applicazione può richiedere genericamente una implementazione di un dato algoritmo senza curarsi di quale provider la fornisca
  - una volta installati, o staticamente o dinamicamente, possono coesistere ed interoperare più CSP, anche di produttori differenti
  - il provider di default si chiama "**Sun**" ed è integrato in JDK

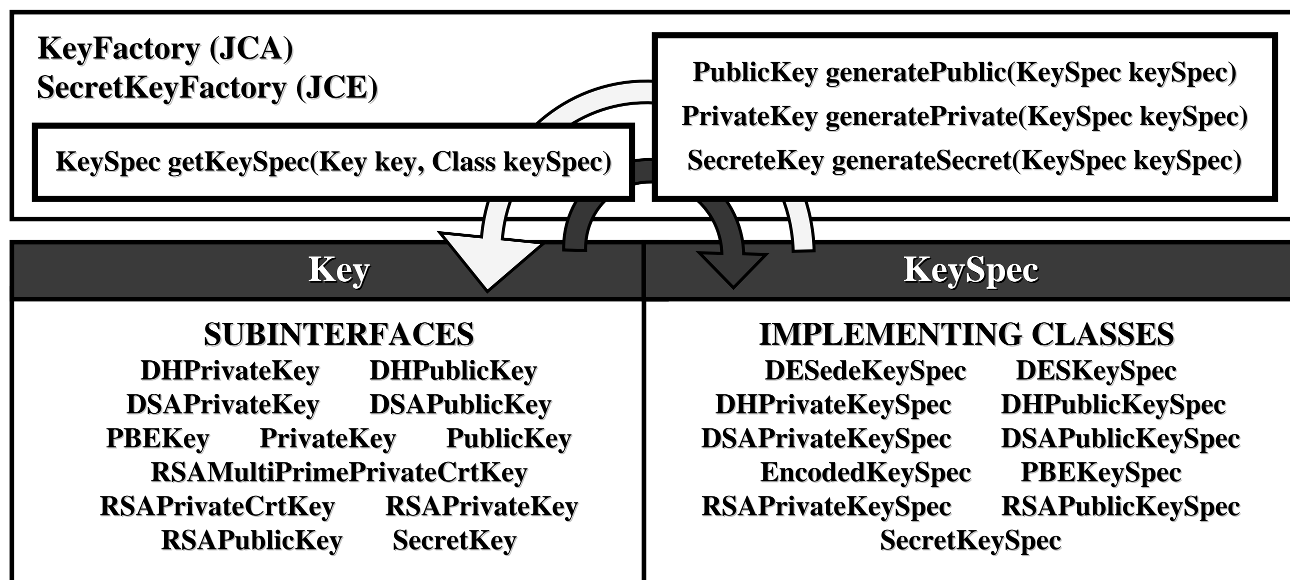
Principali engine classes

- Le seguenti classi di tipo <u>engine</u> sono definite nel package **java.security**:
  - Key – definisce le funzionalità condivise da chiavi di tipo "**opaco**" ("*opaque cryptographic keys*")
  - KeySpec – definisce una chiave di tipo "**trasparente**" ("*transparent representations of the underlying key material*")
  - KeyFactory – rende una chiave di tipo Key o "opaca" o "trasparente"
  - KeyPairGenerator – genera una coppia di chiavi asimmetriche
  - AlgorithmParameters – gestisce i parametri di un algoritmo
  - AlgorithmParameterGenerator – genera i set di parametri di un algoritmo
  - MessageDigest – calcola l'hash (message digest) di dati specifici
  - SecureRandom – genera numeri casuali o pseudo-casuali
  - Signature – appone e verifica la firma digitale
  - CertificateFactory – crea e revoca certificati di chiavi pubbliche
  - KeyStore – crea e gestisce un database (**keystore**) di chiavi e certificati sicuri

N.B. I termini "**opaco**" e "**trasparente**" non hanno lo stesso significato di "**cifrato**" e "**in chiaro**"!

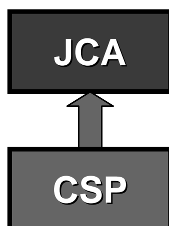
Conversione opaco-trasparente

- *"...An **opaque** key representation is one in which you have no direct access to the key material that constitues a key. In other words: "opaque" gives you <u>limited access</u> to the key - just the three methods defined by the **Key** interface (see below): **getAlgorithm**, **getFormat**, and **getEncoded**. ..."*
- *"...This is in contrast to a **transparent** representation, in which you can access each key material value individually, through one of the "get" methods defined in the corresponding specification class. ..."*



“Sun” Provider

- Il package provider "**Sun**" include:
  - una implementazione dell’algoritmo DSA (Digital Signature Algorithm)
  - una implementazione degli algoritmi MD5 e SHA-1 di MessageDigest
  - un KeyParGenerator per DSA
  - un AlgorithmParameters per DSA
  - un AlgorithmParameterGenerator per DSA
  - una KeyFactory per DSA
  - una implementazione dell’algoritmo proprietario "SHA1PRNG" come SecureRandom (raccomandazioni espresse in IEEE P1363 standard)
  - una CertificateFactory per certificati X.509 e per CRLs
  - una implementazione del KeyStore proprietario chiamato "**JKS**“
- La struttura della JCA è quindi organizzata in <u>due livelli</u>:



- la **JCA** definisce le classi astratte e fornisce solo alcune semplici implementazioni di funzionalità specifiche
- il **CSP** definisce le API complete e fornisce una implementazione di tutte o alcune classi astratte oltre ad eventuali servizi aggiuntivi

Installazione statica di un CSP

- Passi da seguire:
  - Procurarsi l’archivio JAR contenente i file che implementano il CSP ed inserirli nelle cartelle di installazione di JRE:
    $JAVA_HOME\jre\lib\ext\
  - Modificare il file “**java.security**” relativo alle proprietà di sicurezza contenuto in
    $JAVA_HOME\jre\lib\security\

    inserendo la seguente linea di codice

    security.provider.<n>=<ProviderName>

    dove n definisce la priorità con cui la JVM sceglie il provider da utilizzare
- Esempio:

```
.....
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.sun.rsajca.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=org.bouncycastle.jce.provider.BouncyCastleProvider
.....
```

INDIPENDENZA
INTEROPERABILITA’
ESTENDIBILITA’

www.bouncycastle.org

Installazione dinamica di un CSP

- Occorre adoperare principalmente le funzionalità messe a disposizione dalle seguenti classi contenute nel package **java.security**:
  - Provider
    - definisce le caratteristiche di un provider per le Java Security API
    - definisce un nome specifico e un numero di versione
    - permette di implementare:
      - algoritmi (es. DSA, RSA, MD5, SHA-1)
      - metodi per la gestione di chiavi (es. per algorithm-specific keys)
  - Security
    - gestisce tutte le proprietà e i metodi relativi alla sicurezza
    - offre metodi statici per la gestione dei providers:

```
static int addProvider(Provider provider)
static Provider getProvider(String name)
static Provider[] getProviders()
static Set getAlgorithms(String serviceName)
static int insertProviderAt(Provider provider, int position)
.....
```


Implementazione di un CSP

- Tutte le informazioni necessarie sono disponibili in una apposita guida reperibile all'interno della "Java™ 2 SDK, SE Documentation Version 1.4"

  (http://java.sun.com/j2se/1.4.1/docs/guide/security/HowToImplAProvider.html)

How to Implement a Provider for the Java Cryptography Architecture

.....

Steps to Implement and Integrate a Provider

Step 1: Write your Service Implementation Code
Step 2: Give your Provider a Name
Step 3: Write your "Master Class," a subclass of Provider
Step 4: Compile your Code
Step 5: Prepare for Testing: Install the Provider
Step 6: Write and Compile Test Programs
Step 7: Run your Test Programs
Step 8: Document your Provider and its Supported Services
Step 9: Make your Class Files and Documentation Available to Clients

.....

JCA e Java Security Extensions

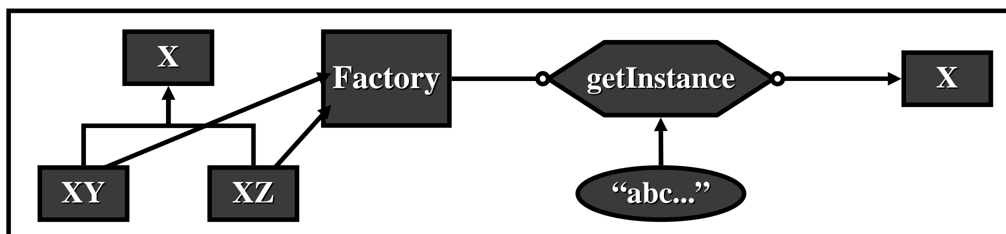
- Le estensioni JCE e JSSE implementano ed estendono le tecniche crittografiche definite dalla JCA fornendo CSP alternativi a “**Sun**”:
  - “**SunJCE**” (crittografia a livello locale)
  - “**SunJSSE**” (crittografia a livello di rete)

- I restanti set di packages offrono invece funzionalità di sicurezza aggiuntive e sono quindi “complementari” alla JCA:
  - JAAS – servizi di autenticazione, autorizzazione e amministrazione
  - CertPath – gestione di certificati e catene di certificati (certification paths)
  - JGSS – meccanismi di comunicazione generici e specifici per Kerberos v5

- Tutti meccanismi di sicurezza realizzati attraverso le Java Security Extensions interagiscono con il Java 1.4 Security Model

Factory Pattern

- Tutte le classi di tipo engine della JCA e delle diverse Security Extensions vengono adoperate in maniera simile tramite un **Factory Pattern:**



- In sostanza per istanziare un oggetto:
  - non si usa la parola chiave **new**...
  - ma un metodo statico: **getIstance(String nomeIstanza)**

- Alcuni esempi:

```
KeyGenerator kg = KeyGenerator.getInstance(“TripleDES”)
KeyPairGenerator kpg = KeyPairGenerator.getInstance(“RSA”)
Signature sig = Signature.getInstance(“MD5WithRSA”)
CertificateFactory cf = CertificateFactory.getInstance(“X.509”)
```

N.B. A provider diversi potrebbero corrispondere nomi diversi!

JCE
Java Cryptography Extension

Caratteristiche

- La Java Cryptography Extension (JCE) ha il compito di fornire una implementazione completa delle funzionalità di cifratura e decifrazione dichiarate dalla JCA
- Soddisfa quindi i principi di progetto della CSP Architecture
- Offre supporto alle seguenti tecniche crittografiche:
  - cifrari simmetrici a blocchi e a flusso
  - cifrari asimmetrici
  - cifrari con password
- Applicabili su:
  - Data
  - I/O Streams
  - Serializable Object
- In più, i meccanismi di:
  - MAC (Message Authentication Code)
  - Key Generation
  - Key Agreement

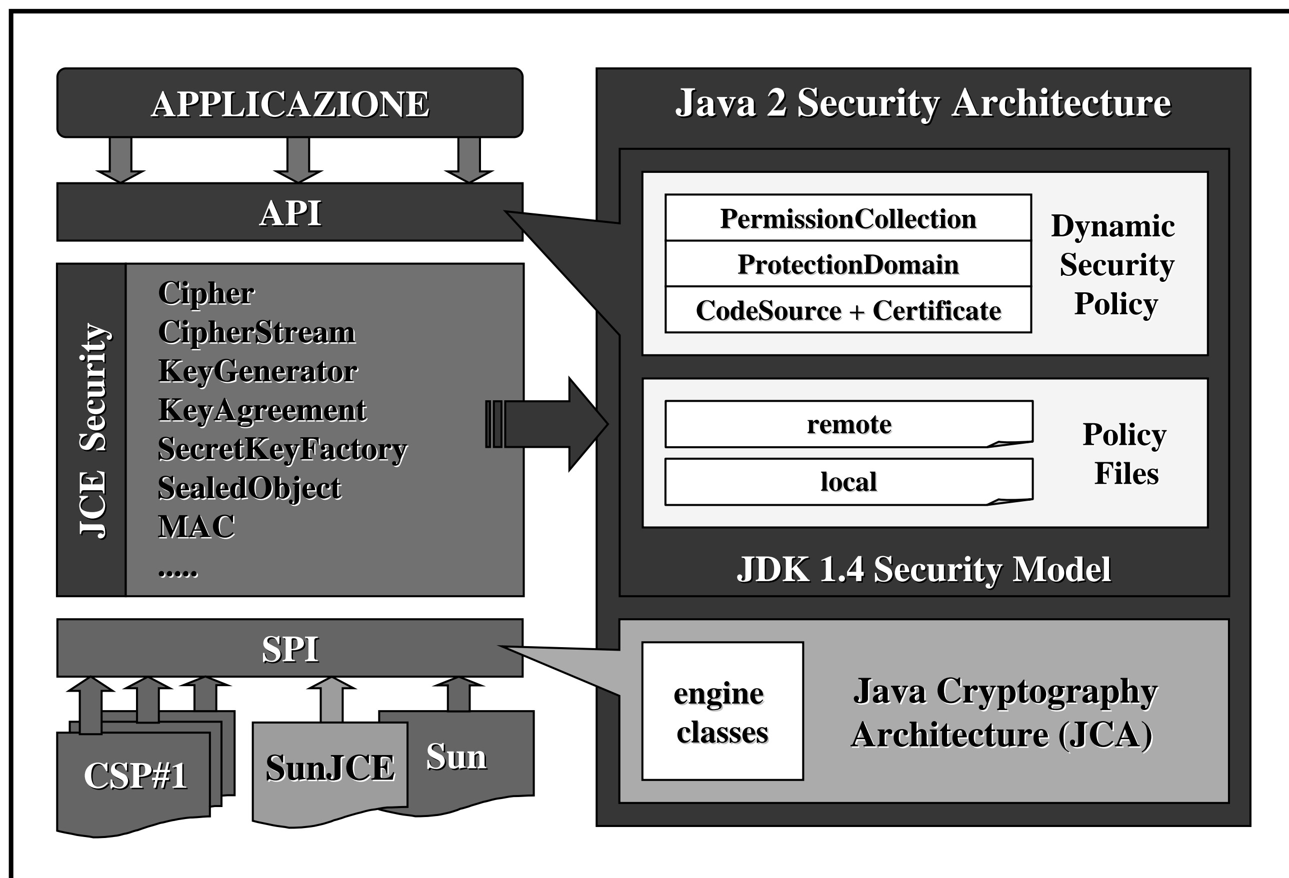
Classi principali

- Le JCE API sono contenute nel set di packages **javax.crypto**
- Alcune classi principali:
  - Cipher – offre funzionalità di cifratura e decifrazione di dati mediante uno specifico algoritmo
  - CipherInputStream / ChiperOutputStream – incapsulano il concetto di canale sicuro, combinano un oggetto Cipher con un InputStream o un OutputStream per gestire automaticamente cifratura e decifrazione durante la comunicazione
  - KeyGenerator – genera chiavi sicure per algoritmi simmetrici e per lo scambio Diffie-Hellmann (DH)
  - SecretKeyFactory – rende una chiave di tipo Key o “opaca” o “trasparente”
  - SealedObject – costruisce oggetti serializzabili che incapsulano il cifrario semplificando la memorizzazione ed il trasferimento di oggetti cifrati
  - KeyAgreement – gestisce i protocolli per concordare una chiave
  - Mac – offre funzionalità di Message Authentication Code (MAC)

“SunJCE” Provider

- Il package provider "**SunJCE**" include:
  - una implementazione degli algoritmi di cifratura DES, TripleDES e Blowfish in modalità ECB, CBC, CFB, OFB e PCBC
  - un KeyGenerator per DES, TripleDES, Blowfish, HMAC-MD5, HMAC-SHA1
  - una implementazione di MD5 con DES-CBC PBE definito in PKCS#5
  - una SecretKeyFactory per una conversione bidirezionale tra oggetti “opaque” DES, TripleDES e PBE e un "key material” trasparente
  - una implementazione dell’algoritmo di Diffie-Hellman di KeyAgreement
  - un KeyGenerator di coppie di chiavi asimmetriche per Diffie-Hellman
  - una SecretKeyFactory per una conversione bidirezionale tra oggetti “opaque” Diffie-Hellman e un " key material” trasparente
  - manager di parametri per DH, DES, TripleDES, Blowfish e PBE
  - una implementazione dello schema di padding definito in PKCS#5
  - una implementazione del keystore proprietario chiamato “**JCEKS**”

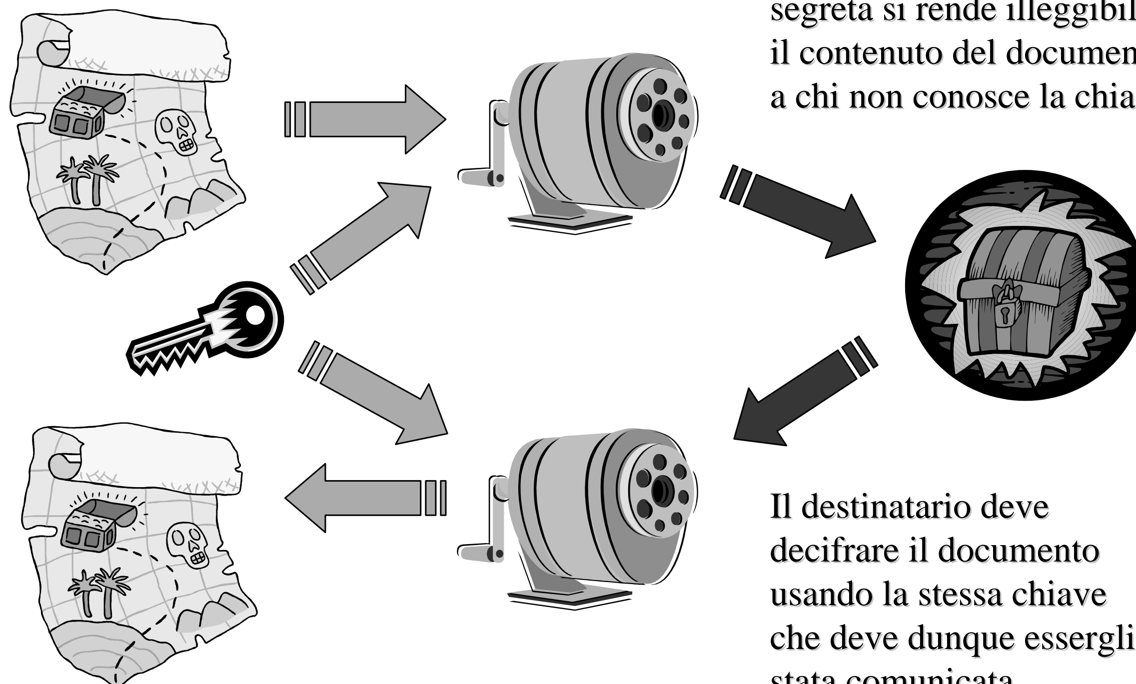
JCE Architecture



Cifratura simmetrica

Si possiede un documento segreto che si vuole trasmettere senza che altri a parte il destinatario lo possano leggere

Attraverso un algoritmo noto e grazie ad una chiave segreta si rende illeggibile il contenuto del documento a chi non conosce la chiave



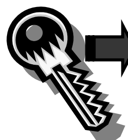
Il destinatario deve decifrare il documento usando la stessa chiave che deve dunque essergli stata comunicata

TripleDES

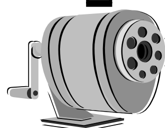
```
import java.security.*;
import javax.crypto.*;

public class TripleDES {
    public static void main (String[] args) {
        String text = "Hello world!";
        if (args.length == 1) text = args[0];
        KeyGenerator keyGenerator = KeyGenerator.getInstance("TripleDES");

        keyGenerator.init(168);

        Key key = keyGenerator.generateKey();

        Cipher cipher = Cipher.getInstance("TripleDES/ECB/PKCS5Padding");
```

Istanzia un generatore di chiavi di tipo "TripleDES"
Se avessi utilizzato il provider di BouncyCastle avrei dovuto usare la stringa "DESede"

Inizializza il generatore di chiavi con la lunghezza in bit della chiave
Per TripleDES si ha sempre 168, ma esistono altri algoritmi che hanno lunghezze di chiavi variabili e/o adoperano un seme casuale per l'inizializzazione



Genera la chiave
Key non viene istanziata con una new

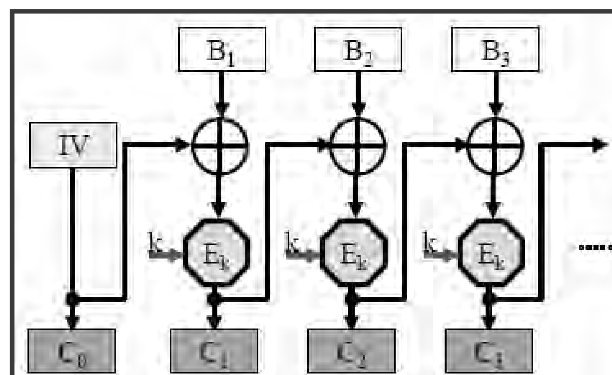


Istanzia il cifrario
Tipo di chiave [TripleDES]
Modalità [ECB]
Padding [PKCS5Padding]

Modalità

- La modalità definisce come un cifrario debba applicare un algoritmo di cifratura
- Si può specificare se un cifrario deve essere a **blocchi** o a **flusso**
- Due tra le modalità più comuni sono:
  - **ECB** [Elettronic Code Book]
    - Lo stesso blocco di testo in chiaro viene sempre cifrato nello stesso blocco di testo cifrato

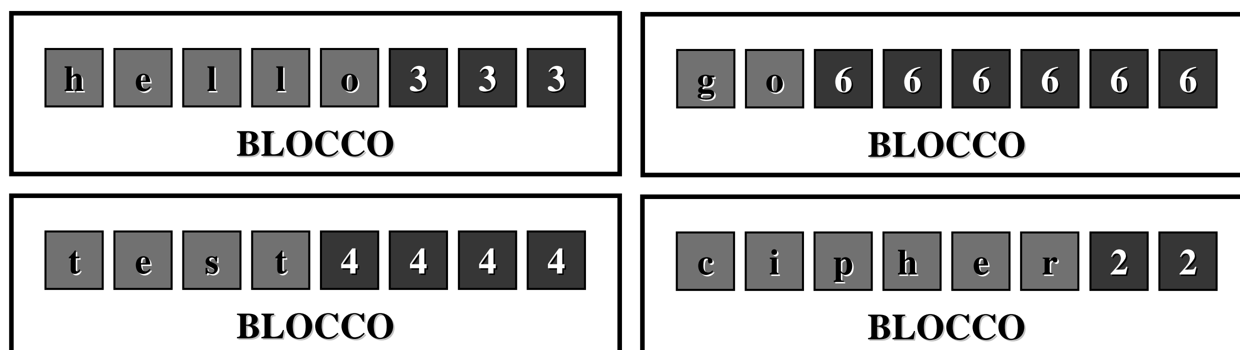
  - **CBC** [Cipher Block Chaining]
    - Ogni blocco cifrato dipende dal relativo blocco in chiaro, da tutti i blocchi precedenti e da un vettore di inizializzazione IV



- Altre modalità sono:
  - **CFB** [Cipher Feedback]
  - **OFB** [Output Feedback]

Padding

- I cifrari a blocchi operano su blocchi di dati ma difficilmente i dati in chiaro avranno esattamente una dimensione pari ad un multiplo del blocco
- Prima di cifrare occorre dunque aggiungere il **padding** (spaziatura)
- Il padding PKCS#5 (Public Key Criptography Standard) è il più utilizzato per la cifratura simmetrica:
  - ai byte che mancano per riempire un blocco viene assegnato un numero che equivale alla quantità di byte mancanti
- Alcuni esempi:



TripleDES

```
......
Cipher cipher = Cipher.getInstance("TripleDES/ECB/PKCS5Padding");

cipher.init(Cipher.ENCRYPT_MODE, key);

byte[ ] plaintext = text.getBytes("UTF8");

System.out.println("\nPLAINTEXT:");
for (int i=0;i<plaintext.length;i++)
        System.out.print(plaintext[i]+" ");

byte[ ] ciphertext = cipher.doFinal(plaintext);

System.out.println("\n\nCIPHERTEXT:");
for (int i=0;i<ciphertext.length;i++)
        System.out.print(ciphertext[i]+" ");

cipher.init(Cipher.DECRYPT_MODE, key);

byte[ ] decryptedText = cipher.doFinal(ciphertext);

String output = new String(decryptedText,"UTF8");
System.out.println("\n\nDECRYPTED TEXT:\n"+output);
......
```

Inizializza il cifrario per la modalità di cifratura

Converte una stringa in un array di byte specificando la codifica
Evita problemi di conversione di array di byte a stringa in caso di mittente e destinatario su piattaforme differenti (default: codifica della macchina sottostante)

Cifra i dati
Prima di cifrare si possono inserire altri dati con il metodo update(byte[]) (ricordarsi di specificare la codifica)

Inizializza il cifrario per la modalità di decifrazione
Viene usata la stessa chiave

Decifra i dati

Eseguire l’esempio

- Comandi per la compilazione del sorgente e l’esecuzione:

```
javac TripleDES.java
java TripleDES [“testo da cifrare”]
```

- L’output che dovrebbe apparire:

```
Plaintext:
72 101 108 108 111 32 119 111 114 108 100 33

Ciphertext:
23 28 -14 60 -14 45 -6 23 12 53 71 58 123 90 -108 -57

DecryptedText:
Hello world!
```


Blowfish e Rijndael

- Cambiare cifrario risulta davvero molto semplice...
  - Ad esempio per passare a Blowfish basta cambiare poche righe di codice:

```
.....
KeyGenerator keyGenerator = KeyGenerator.getInstance(“Blowfish");
keyGenerator.init(128);
.....
Cipher cipher = Cipher.getInstance(“Blowfish/ECB/PKCS5Padding");
.....
```

  - Mentre per cifrare con Rijndael:

```
.....
Cipher cipher = Cipher.getInstance(“Rijndael/CBC/PKCS5Padding");

SecureRandom random = new SecureRandom();
byte[] iv = new byte[16];
random.nextBytes(iv);
IvParameterSpec spec = new IvParameterSpec(iv);

cipher.init(Cipher.ENCRYPT_MODE, key, spec);
.....
```

Il cifrario viene istanziato in modalità CBC

La classe SecureRandom permette di generare i byte random che permettono a loro volta la creazione del seme grazie alla classe IvParameterSpec (javax.crypto.spec)
Il numero di byte deve corrispondere alla dimensione del blocco

Inizializza il cifrario passando anche il seme creato con IvParameterSpec

Eccezioni

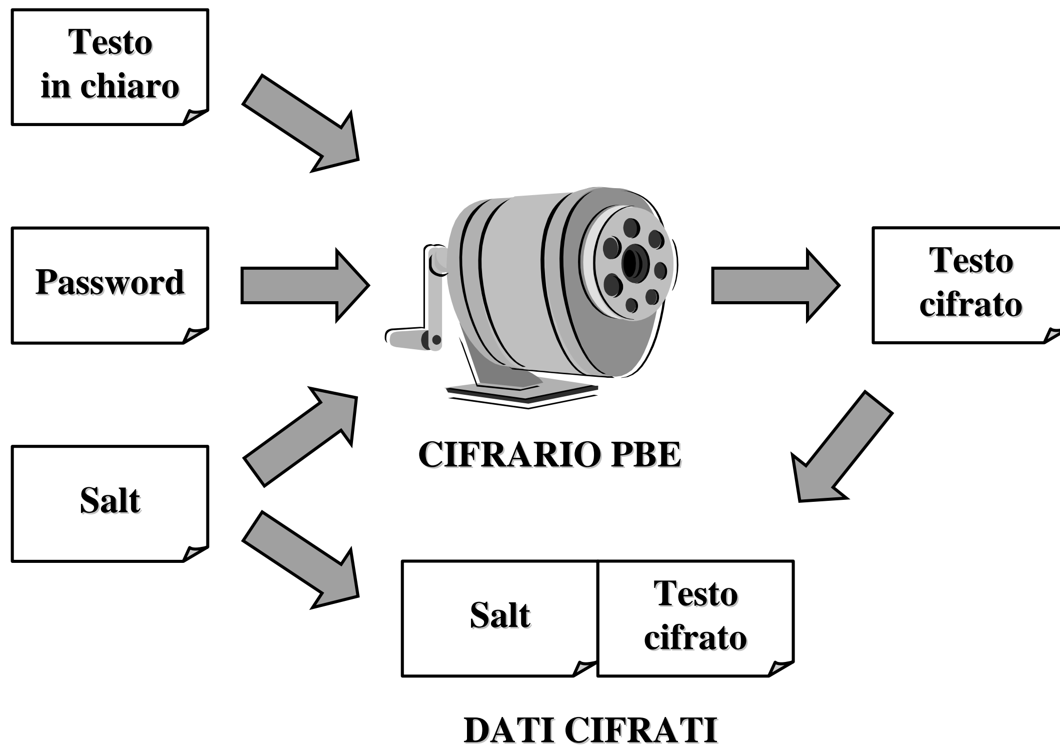
- Il codice visto così com'è non funziona!
  - Occorre infatti gestire le eccezioni...

```
catch (NoSuchAlgorithmException e1) {
        System.out.println("Algoritmo non supportato..."); ... }
catch (InvalidAlgorithmParameterException e2) {
        System.out.println("Parametro non valido"); ... }
catch (NoSuchProviderException e2) {
        System.out.println("Algoritmo non supportato dal provider specificato..."); ... }
catch (NoSuchPaddingException e3) {
        System.out.println("Padding non supportato..."); ... }
catch (BadPaddingException e4) {
        System.out.println("Padding non riuscito..."); ... }
catch (InvalidKeyException e5) {
        System.out.println("Chiave non valida..."); ... }
catch (IllegalBlockSizeException e6) {
        System.out.println("Dimensione blocco non corretta..."); ... }
catch (UnsupportedEncodingException e7) {
        System.out.println("Codifica non supportata..."); ... }
```


Password-Based Encryption

- Algoritmo meno potente di TripleDES o Blowfish
  - tali algoritmi usano chiavi fino a **448** bit
  - la password di un utente medio è di circa 6 caratteri e quindi di **48** bit
  - come password si tende ad usare parole con un determinato significato → keyspace davvero limitato
- Le password sono soggette agli attacchi con dizionario, in difesa dei quali si possono adottare due tecniche di difesa:
  - **Salting**
    - Consiste nell'aggiungere alla password un insieme di bit casuali per ampliarne il keyspace
  - **Conteggi di ripetizione**
    - Consiste nell'effettuare molte volte un'operazione sulla password per ottenere la chiave per il cifrario PBE
    - Ad esempio, applicando alla password 1000 volte un algoritmo di hash per un intruso sarà 1000 volte più difficile rompere il cifrario

Cifratura



Cifratura

```
import java.security.*;          import java.security.spec.*;
import javax.crypto.*;           import javax.crypto.spec.*;
import java.util.*;
public class PBE {
  private final int ITERATIONS = 1000;
  public String encrypt(char[] password, String plaintext) {
    byte[ ] salt = new byte[8];
    Random random = new Random();
    random.nextBytes(salt);
    PBEKeySpec keySpec = new PBEKeySpec(password);
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
    SecretKey key = keyFactory.generateSecret(keySpec);
    PBEParameterSpec paramSpec = new PBEParameterSpec(salt, ITERATIONS);
    Cipher cipher = Cipher.getInstance("PBEWithMD5AndDES");
    cipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
    byte[ ] ciphertext = cipher.doFinal(plaintext.getBytes());
    String saltString = new String(salt);
    String ciphertextString = new String(ciphertext);
    return saltString+ciphertextString;
  } .....
```

Usato per i conteggi di ripetizione

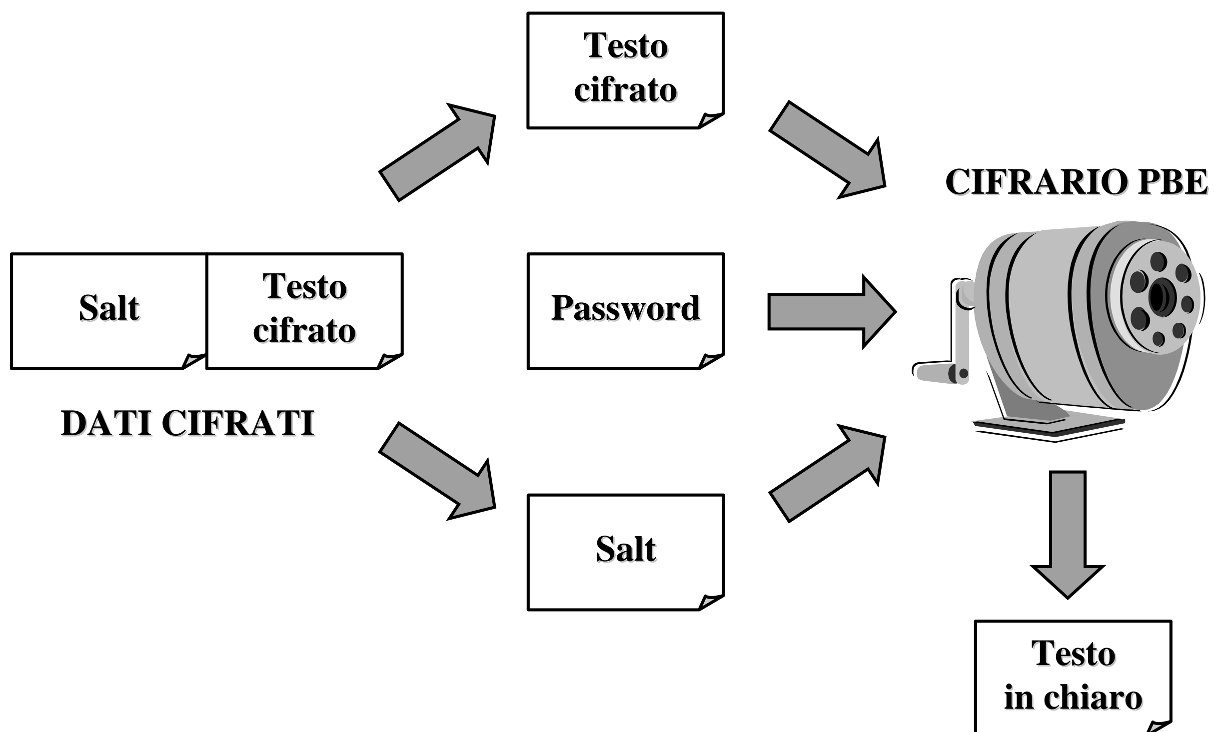
Generazione di un salt casuale

La classe PBEKeySpec serve per creare una chiave basata su una password usando una istanza di SecretKeyFactory

PBEParameterSpec contiene il salt e il numero di iterazioni
Una sua istanza deve essere passata al cifrario PBE

Concatenazione del salt con i dati cifrati

Decifratura



Decifratura

```
.....
public static String decrypt(char[] password, String input) {

    String saltString = input.substring(0,8);
    String ciphertextString = input.substring(8,input.length());
    byte[ ] salt = saltString.getBytes();
    byte[ ] ciphertext = ciphertextString.getBytes();

    PBEKeySpec keySpec = new PBEKeySpec(password);
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
    SecretKey key = keyFactory.generateSecret(keySpec);
    PBEParameterSpec paramSpec = new PBEParameterSpec(salt,ITERATIONS);
    Cipher cipher = Cipher.getInstance("PBEWithMD5AndDES");
    cipher.init(Cipher.DECRYPT_MODE, key, paramSpec);

    byte[ ] plaintextArray = cipher.doFinal(ciphertext);
    return new String(plaintextArray);
} .....
```

Estrazione del salt e del testo cifrato

Esattamente come prima, ma il cifrario è ora in DECRYPT_MODE

Cifratura della chiave

- Memorizzare la chiave su floppy o smart card è scomodo perchè servono da supporto mezzi fisici esterni
- Supponiamo allora di utilizzare il disco fisso:
  - la memorizzazione su File System riduce la sicurezza delle chiavi
  - occorre quindi cifrare la chiave, ad esempio con PBE
  - conviene inoltre proteggere ulteriormente la chiave ad esempio impostando i permessi di accesso al file
- Cifratura con PBE

```
byte[] keyBytes = myKey.getEncoded();
cipher.init(Cipher.ENCRYPT_MODE, passwordKey, paramSpec);
byte[] encrytedKeyByte = cipher.doFinal(keyBytes);
```

Una volta in possesso di una chiave con getEncoded() si restituisce un array di byte, che è possibile cifrare con PBE

- Decifratura con PBE

```
cipher.init(Cipher.DECRYPT_MODE, passwordKey, paramSpec);
byte[ ] keyBytes = cipher.doFinal(encrytedKeyByte);
SecretKeySpec mykey = new SecretKeySpec(keyBytes, "Blowfish");
```

Riporta la chiave ad un formato di più alto livello
Occorre specificare il tipo di chiave

Incapsulamento ed estrazione della chiave

- Alcuni provider che implementano JCE forniscono un più comodo mezzo per la cifratura della chiave (che evita la conversione "byte[ ]" ⟺ "Key")
- Occorre un cifrario PBE
- E' possibile incapsulare una chiave segreta come segue:

```
cipher.init(Cipher.WRAP_MODE, passwordkey, paramSpec);
byte[] encryptedKeyBytes = cipher.wrap(secretKey);
```

Occorre inizializzare il cifrario in WRAP_MODE invece che in ENCRYPT_MODE
Si usa il metodo wrap() con argomento una Key

- Ed estrarre poi la chiave nel seguente modo:

```
cipher.init(Cipher.UNWRAP_MODE, passwordkey, paramSpec);
Key key =  cipher.unwrap(encryptedKeyBytes, "Blowfish", Cipher.SECRET_KEY);
```

Occorre inizializzare il cifrario in UNWRAP_MODE invece che in DECRYPT_MODE
Il metodo unwrap() richiede l'algoritmo della chiave incapsulata ed il tipo di chiave SECRET_KEY. Restituisce una chiave di tipo Key

CipherStreams

- CipherInputStream e CipherOutputStream incapsulano il concetto di canale sicuro
- Combinano un oggetto Cipher con un InputStream o un OutputStream per gestire automaticamente cifratura e decifrazione durante la comunicazione
- Esempio di costruzione:

```
FileOutputStream output = new FileOutputStream("cipherText_FileName") ;
CipherOuputStream cipherOutput = new CipherOutputStream (ouput,cipher);
FileInputStream input = new FileInputStream("cipherText_FileName");
CipherInputStream cipherInput = new CipherInputStream(input, cipher);
```

CipherOutputStream è adoperato per la cifratura
Cifrario in ENCRYPT_MODE
CipherInputStream serve invece per la decifrazione
Cifrario in DECRYPT_MODE

- Esempio di utilizzo per la cifratura:

```
...
int r = 0;
while (r = input.read() != -1) { cipherOutput.write(r); }
cipherOuput.close();
ouput.close();
input.close();
...
```

Ogni carattere in chiaro viene automaticamente cifrato
Nell'esempio "input" è un normale InputStream

SealedObject

- I SealedObject sono oggetti cifrati che incapsulano il cifrario
- Gli oggetti "chiusi" (tramite JCE) possono essere utili per <u>memorizzare</u> e <u>trasportare</u> una versione cifrata di un oggetto
- L'oggetto deve essere serializzabile (!)

```
import java.io.*;
import javax.crypto.*;
import java.security.*;
public class SealedObjectExample {
  public static void main (String[ ] args) {
   String secretMessage = "Ci vediamo domani alle 15 al bar Roma";
   KeyGenerator keyGenerator = KeyGenerator.getInstance("Blowfish");
   Key key = keyGenerator.generateKey();
   Cipher cipher = Cipher.getInstance("Blowfish/ECB/PKCS5Padding");
   cipher.init(Cipher.ENCRYPT_MODE, key);
   SealedObject so = new SealedObject(secretMessage, cipher);
   String decryptedMessage = (String)so.getObject(key);
   System.out.println("\nMessaggio decifrato: "+decryptedMessage);
   .....
```

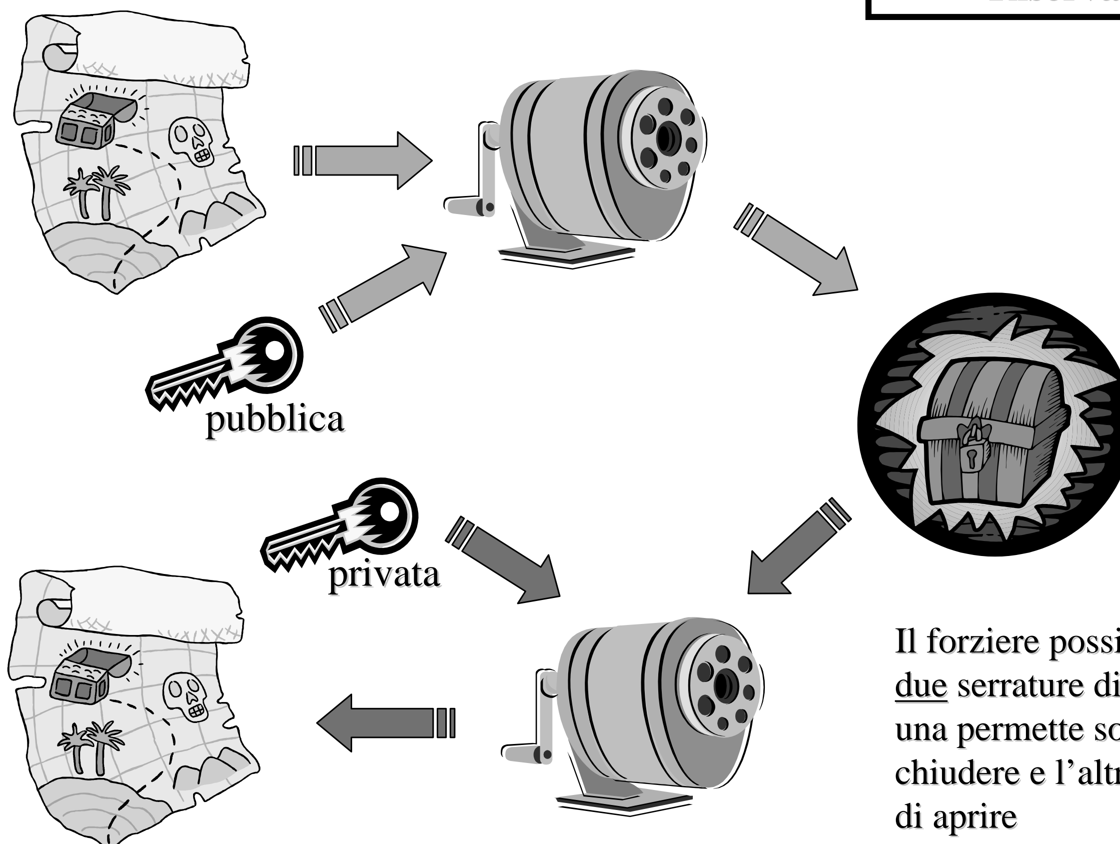
Il costruttore di SealedObject chiede che gli venga passato l'oggetto da cifrare ed il cifrario con cui chiuderlo

Per la decifrazione non occorre settare il cifrario, è sufficiente la chiave
Occorre effettuare un cast

Cifratura asimmetrica

- La cifratura **SIMMETRICA** è un ottimo strumento per la sicurezza
- Il problema della cifratura simmetrica sta nel fatto che occorre avere una chiave condivisa e quindi bisogna scambiarsi questa informazione
- Questa operazione è però molto difficile da effettuarsi in modo che nessun altro possa scoprire la chiave
- La soluzione a questo problema è agevolmente risolto attraverso la cifratura a chiave **ASIMMETRICA**

Cifratura asimmetrica



Il forziere possiede ora due serrature diverse: una permette solo di chiudere e l'altra solo di aprire

Modalità

- Nei cifrari asimmetrici si usa quasi sempre ECB
- Tipicamente i cifrari asimmetrici sono utilizzati per cifrare un singolo blocco in chiaro
- Generalmente le dimensioni di quel blocco sono grandi quasi come quelle della chiave
- Quando è necessario cifrare una quantità maggiore di dati si ricorre generalmente alla cifratura a chiave di sessione che vedremo in seguito

Padding

- Nella cifratura asimmetrica non si usa PKCS#5
- Gli standard per cifrare con RSA sono invece PKCS#1 e OAEP (Optimal Asymmetric Encryption Padding)
- Di seguito non sarà illustrato il funzionamento di tali forme di padding
- Per chi fosse interessato : www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html

RSA

Occorre un provider che supporti tale algoritmo
Es. **BouncyCastle** (←)
In J2SE1.4 RSA è supportato solo per le operazioni di firma e verifica

```
import javax.crypto.*;
import java.security.*;
…..
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");

keyPairGenerator.initialize(1024);

KeyPair keyPair = keyPairGenerator.genKeyPair();

Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");

cipher.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());

byte[] messageBytes = new String("Combinazione:1234").getBytes("UTF8");
byte[] cipherText = cipher.doFinal(messageBytes);

cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());

byte[] decryptedMessageBytes = cipher.doFinal(cipherText);
System.out.println(new String(decryptedMessageBytes,"UTF8"));
.....
```

Questa volta occorre una coppia di chiavi

Per la cifratura occorre inizializzare il cifrario con una PublicKey

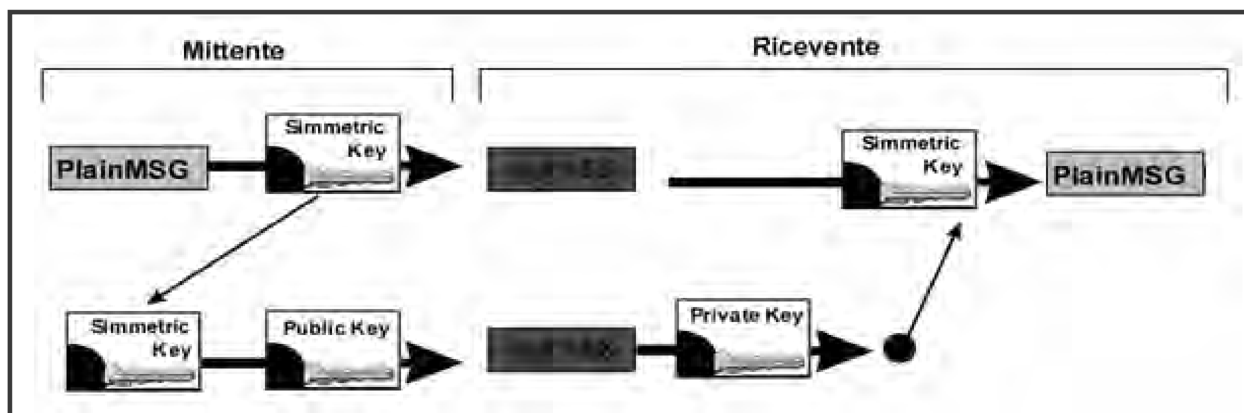
Per la decifrazione occorre inizializzare il cifrario con una PrivateKey

Cifratura a chiave di sessione

La cifratura asimmetrica è **1000** volte più lenta di quella simmetrica

Per gestire questa inefficienza in genere si utilizza la cifratura a **CHIAVE DI SESSIONE**

Cifratura a chiave di sessione



1. Il messaggio in chiaro viene cifrato con una chiave simmetrica
2. La chiave simmetrica viene cifrata con la chiave pubblica del destinatario
3. Il destinatario decifra la chiave simmetrica con la sua chiave privata
4. Il destinatario può decifrare il messaggio usando la chiave simmetrica ottenuta

Cifrando la chiave simmetrica, la cifratura asimmetrica coinvolge poco testo in chiaro risultando in questo modo efficiente. La chiave simmetrica inoltre viene cambiata ad ogni sessione garantendo maggiore sicurezza (da qui il nome di **chiave di sessione**)

Cifratura di una chiave simmetrica

```
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;
.....
KeyGenerator keyGenerator = KeyGenerator.getInstance("Blowfish");
keyGenerator.init(128);
Key blowfishKey = keyGenerator.generateKey();

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
keyPairGenerator.initialize(1024);
KeyPair keyPair = keyPairGenerator.genKeyPair();

Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
cipher.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());

byte[] blowfishKeyBytes = blowfishKey.getEncoded();

byte[] cipherText = cipher.doFinal(blowfishKeyBytes);
cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
byte[] decryptedKeyBytes = cipher.doFinal(cipherText);

SecretKey blowfishKey = new SecretKeySpec(decryptedKeyBytes,"Blowfish");
.....
```

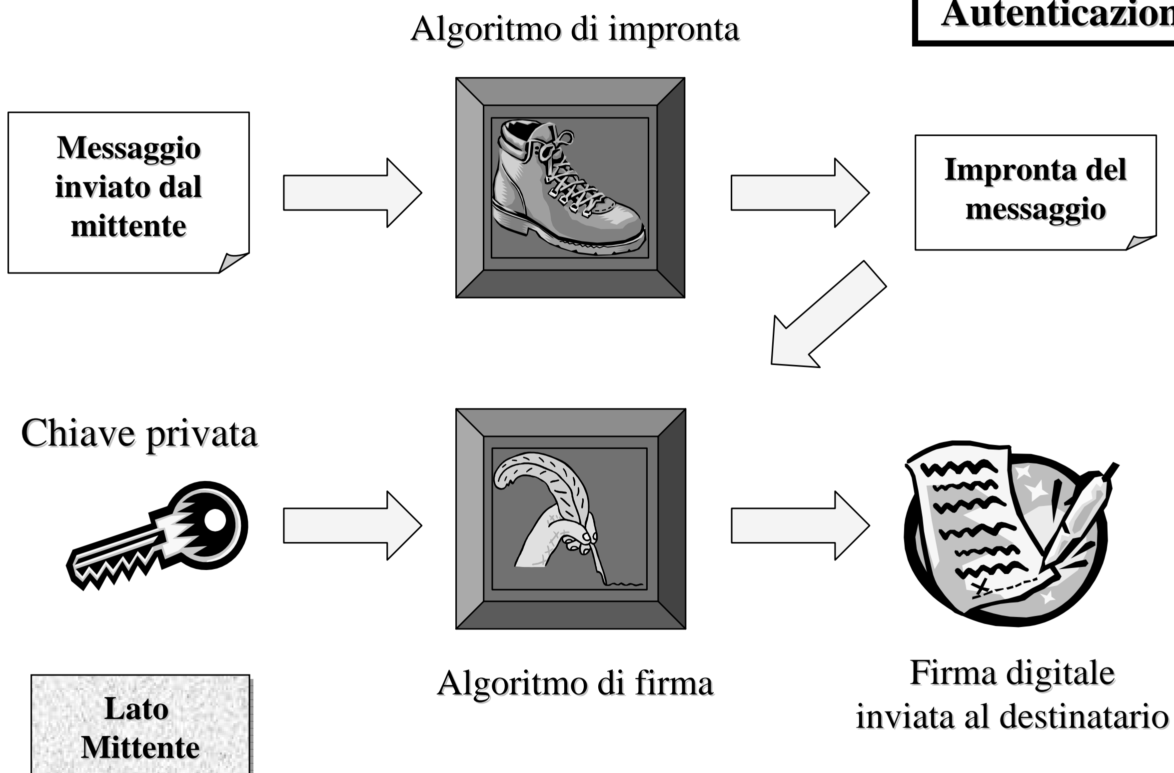
Generazione della chiave di sessione

Generazione di una coppia di chiavi RSA

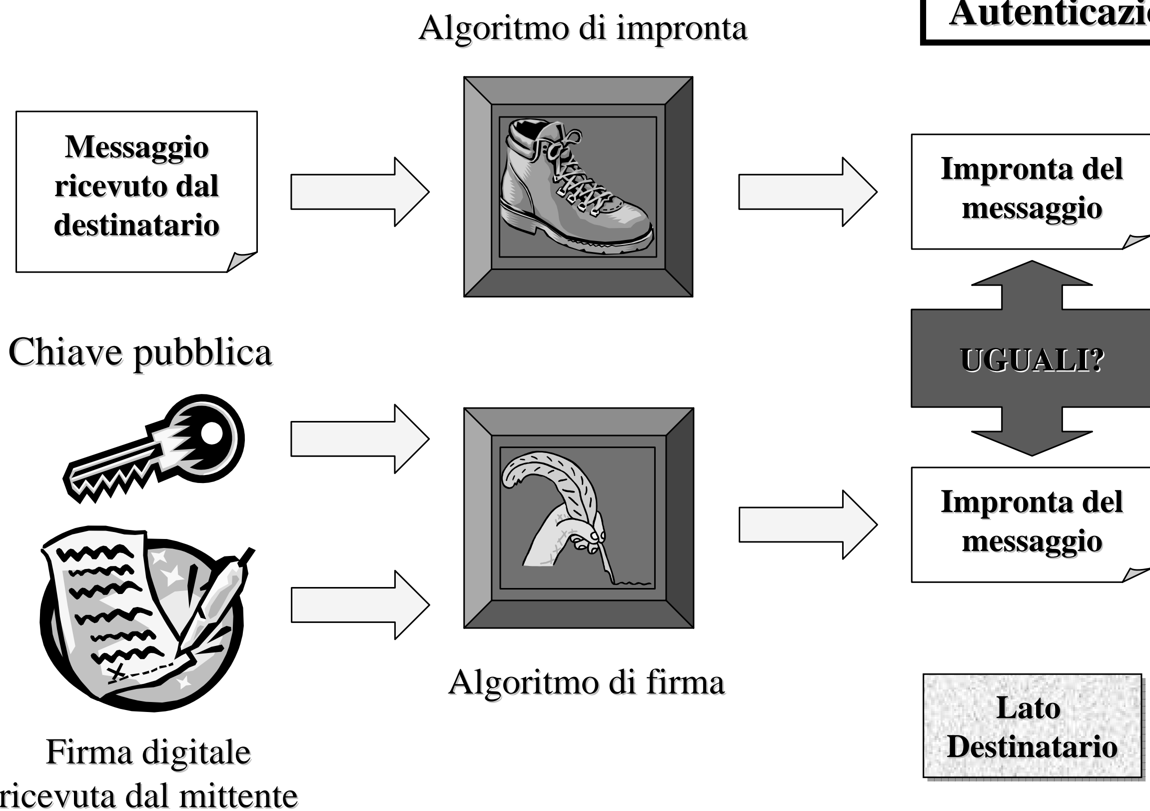
Trasformazione della chiave nel formato richiesto per l'operazione di cifratura

Riconversione dei byte in chiave

Firme digitali (1)



Firme digitali (2)



Firmare e verificare

```
import java.security.Signature;              import java.security.SignatureException;
import java.security.KeyPair;                import java.security.KeyPairGenerator;
.....
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(1024);
KeyPair keyPair = kpg.genKeyPair();
byte[] messaggio = new String("Non esistono più le mezze stagioni").getBytes("UTF8");

Signature sig = Signature.getInstance("MD5WithRSA");

sig.initSign(keyPair.getPrivate());
sig.update(messaggio);
byte[] signatureBytes = sig.sign();
.....
sig.initVerify(keyPair.getPublic());
sig.update(messaggio);

boolean verified = false;
try { verified = sig.verify(signatureBytes); }
catch (SignatureException se){ System.out.println("La firma ha un formato non valido!"); }
if (verified) { … } else { … }
.....
```

Per firmare occorre istanziare un oggetto Signature

Per firmare si inizializza con la chiave privata
update() passa i dati da firmare
sign() restituisce i byte della firma digitale

Per la verifica della firma si inizializza con la chiave pubblica
update() passa i dati da verificare

SignatureException è lanciata, ad esempio, in caso di perdita di un byte e non perchè la verifica ha dato esito negativo

RSA e DSA a confronto

- Firmare con RSA significa cifrare con la chiave privata e decifrare con la pubblica
- Questa operazione non nasconde i dati (perchè decifrabili attraverso la chiave pubblica) ma prova l'identità del firmatario
- Il DSA (Digital Signature Algorithm) funziona come RSA per la firma ma non può essere usato per cifrare
- Il DSA è più veloce nel generare le firme mentre RSA nel convalidarle

Una firma viene convalidata più spesso di quanto non venga generata
→ RSA risulta quindi più veloce per la maggior parte delle applicazioni

Identità della chiave pubblica

- Per convalidare una firma occorre una chiave pubblica
- Ma come essere certi che la chiave pubblica è autentica?
- Serve un qualche mezzo per averne la certezza

I CERTIFICATI sono il tentativo di risolvere questo problema attribuendo l'identità ad una chiave pubblica in modo inconfutabile

- Java offre funzionalità di gestione dei certificati attraverso un set di packages appositamente dedicato
- Tale componente prende il nome di Java CertPath

JSSE
Java Secure Socket Extension

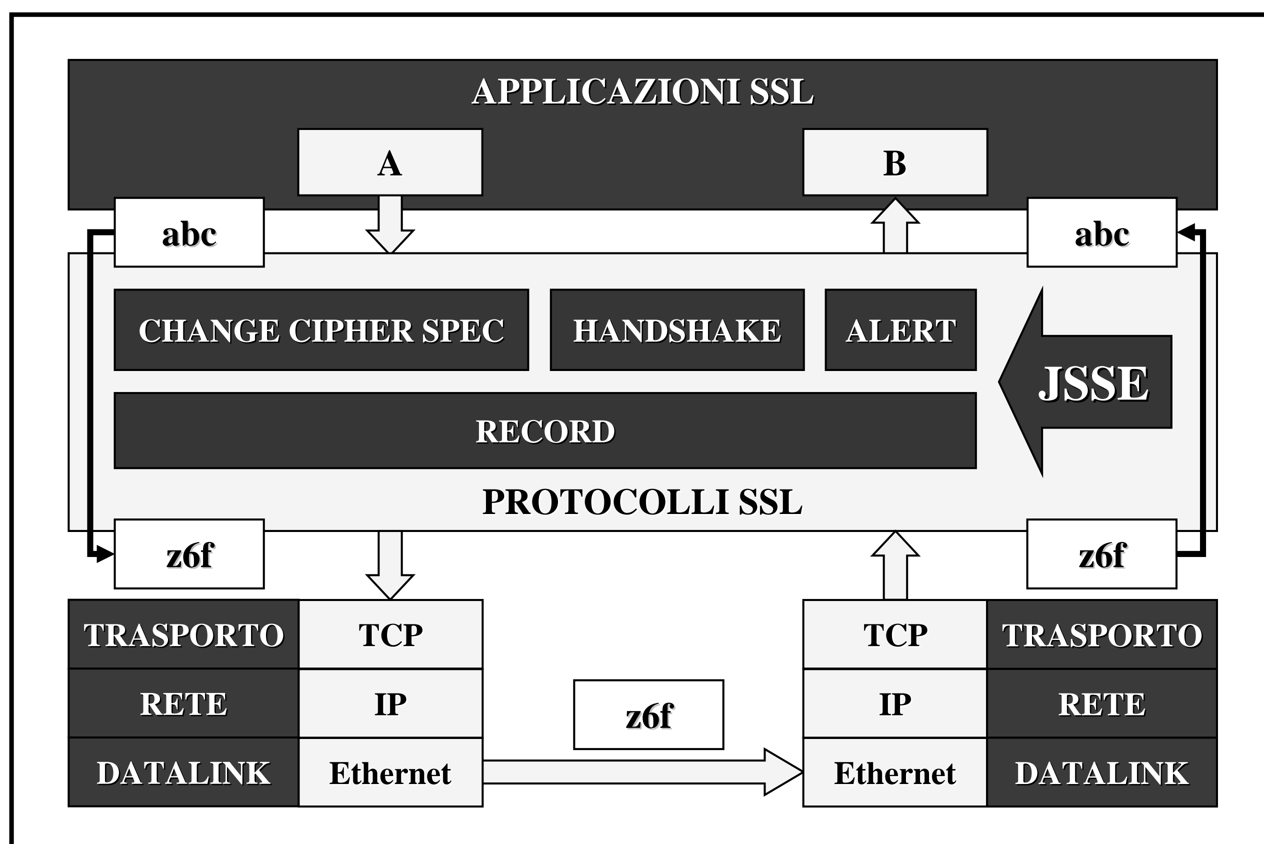
Caratteristiche

- La Java Secure Socket Extensions (JSSE) offre:
  - funzionalità di autenticazione
  - protezione dell'integrità dei dati
  - protezione della riservatezza dei dati
- Mediante i protocolli per comunicazioni sicure:
  - SSL (Secure Socket Layer) v2.0 e v3.0
  - TLS (Transport Layer Security) v1.0
- Mentre JCE opera su specifici dati locali, JSSE adotta una differente astrazione applicando meccanismi crittografici a livello di rete
- Le JSSE API contenute in **javax.net** e **javax.net.ssl** estendono:
  - **javax.crypto** (JCE)
  - **java.security** (JCA)
  - **java.net**

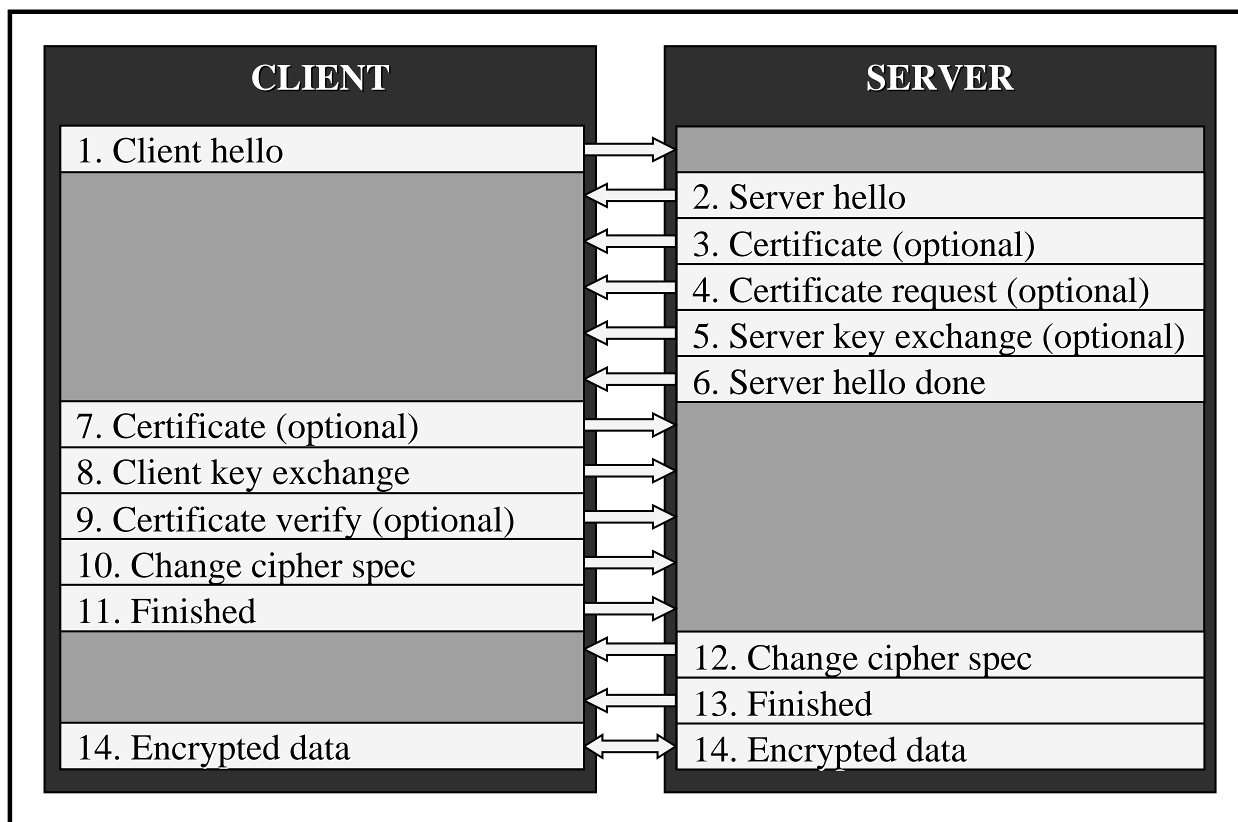
Caratteristiche

- Le funzionalità crittografiche di JSSE sono implementate dal provider "**SunJSSE**"
- Il protocollo di applicazione più adoperato con JSSE è HTTP (Hyper Text Transfer Protocol) che con l'uso di SSL assume la dicitura HTTPS
- Molti altri protocolli possono usufruire di JSSE, come NNTP (Net News Transfer Protocol), Telnet, LDAP (Lightweight Directory Access Protocol), IMAP (Interactive Message Access Protocol) ed FTP (File Transfer Protocol)
- Novità di JSSE nella versione integrata:
  - Il package **javax.security.cert** è presente solo per compatibilità con le applicazioni realizzate con JSSE opzionale. Ad esso è da preferire il package **java.security.cert** di CertPath, integrato in J2SE a partire dalla versione 1.4
  - JSSE può fare uso del provider "**SunJCE**" per operazioni di cifratura e decifrazione

Architettura SSL



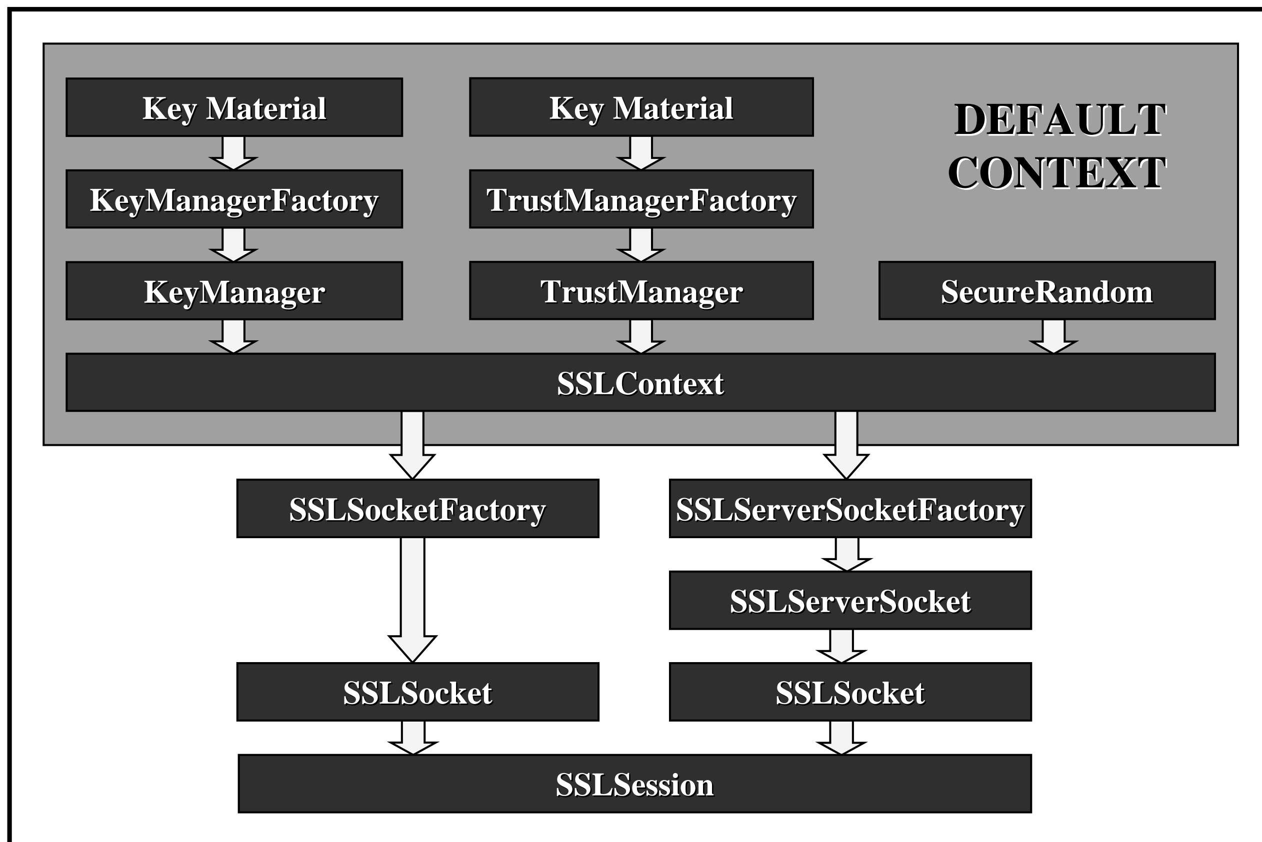
Protocollo di Handshake



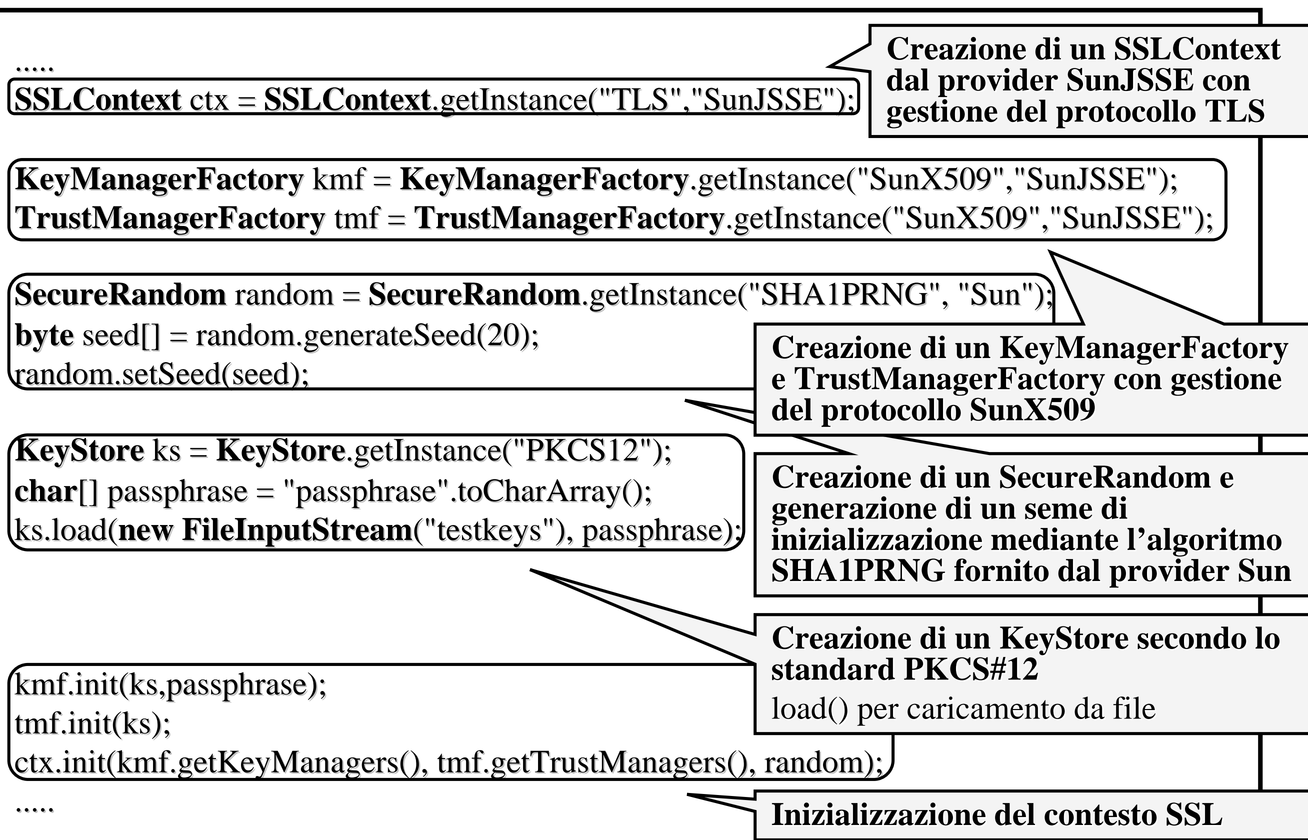
Classi principali

- Factories SSLServerSocketFactory e SSLSocketFactory per la creazione di Socket, ServerSocket, SSLSocket, SSLServerSocket con incapsulamento delle caratteristiche di costruzione e configurazione (socket factory framework)
- Classe SSLContext che agisce da factory per le socket factories e realizza il concetto di contesto sicuro
- Interfacce KeyManager e TrustManager (incluse specifiche X.509) per la gestione delle chiavi e delle decisioni inerenti la sicurezza (keystore e truststore)
- Factories KeyManagerFactory e TrustManagerFactory
- SSLSession per la gestione server di sessioni SSL residenti in memoria
- Connessioni sicure con HttpsURLConnection

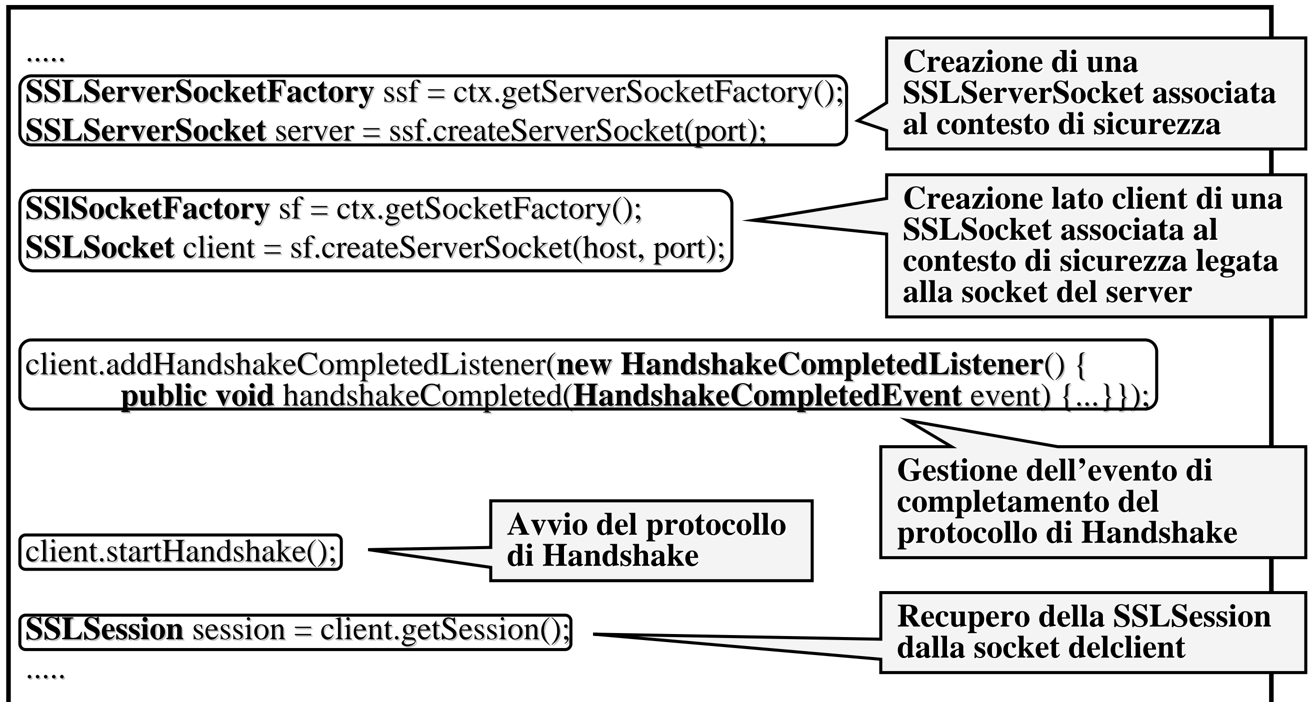
Relationship



Creazione di una SSLContext



Creazione di una SSLSession



SunJSSE Provider

- Funzionalità crittografiche implementate da JSSE:

ALGORITMO CRITTOGRAFICO	PROCESSO CRITTOGRAFICO	LUNGHEZZA CHIAVE (BITS)
RSA	**autenticazione e scambio della chiave**	**2048 (autenticazione) 2048 (scambio della chiave) 512 (scambio della chiave)**
RC4	**cifratura**	**128 128 (40 effettivi)**
DES	**cifratura**	**64 (56 effettivi) 64 (40 effettivi)**
TripleDES	**cifratura**	**192 (112 effettivi)**
Diffie-Hellman	**scambio della chiave**	**1024 512**
DSA	**autenticazione**	**1024**

SunJSSE Provider

- Engine classes della JCA implementate dal provider della JSSE:

ENGINE CLASS	ALGORITMO - PROTOCOLLO
KeyFactory	"RSA"
KeyPairGenerator	"RSA"
KeyStore	"PKCS12"
Signature	"MD2withRSA"
Signature	"MD5withRSA"
Signature	"SHA1withRSA"
KeyManagerFactory	"SunX509"
TrustManagerFactory	"SunX509" - "SunPKIX"
SSLContext	"SSL"
SSLContext	"SSLv3"
SSLContext	"TLS"
SSLContext	"TLSv1"

Cipher Suites

- Cipher suites supportate da SunJSSE in ordine di preferenza per default:

CIPHER SUITES	NUOVE IN J2SEv1.4
SSL_RSA_WITH_RC4_128_MD5	
SSL_RSA_WITH_RC4_128_SHA	
TLS_RSA_WITH_AES_128_CBC_SHA	X
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	X
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	X
SSL_RSA_WITH_3DES_EDE_CBC_SHA	
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	X
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	
SSL_RSA_WITH_DES_CBC_SHA	
SSL_DHE_RSA_WITH_DES_CBC_SHA	X
SSL_DHE_DSS_WITH_DES_CBC_SHA	
SSL_RSA_EXPORT_WITH_RC4_40_MD5	
SSL_DSS_EXPORT_WITH_DES40_CBC_SHA	X
.....	

JAAS
Java Authentication and Authorization Service

Caratteristiche

- Java Authentication and Authorization Service (JAAS) offre due servizi:
  - autenticazione - identificazione sicura dell'utente
  - autorizzazione - verifica dei permessi necessari per le operazioni richieste
- Attraverso:
  - configurazione dinamica dei moduli di login
  - operazioni di callback
  - controllo degli accessi
  - distribuzione dei permessi
- Il framework di autenticazione è progettato per essere "pluggable" in base al modello PAM (Pluggable Authentication Module), che rende indipendenti le applicazioni dalle tecnologie di autenticazione sottostanti come:
  - verifica di username e password
  - verifica di caratteristiche biometriche (voce, impronte digitali, ...)
- Inoltre JAAS supporta lo stacking dei moduli di autenticazione
- Il componente JAAS per le autorizzazioni opera in congiunzione con il modello Java per il controllo degli accessi a risorse sensibili (Java Security Framework)

Classi principali

- Le JAAS APIs sono contenute nel set di packages **javax.security.auth**:

IDENTIFICAZIONE

- Subject – sorgente di richiesta di autenticazione nell'accesso ad una risorsa
- Principal – interfaccia che rappresenta l'identità di un Subject se l'autenticazione ha successo
- Refreshable / Destroyable – interfacce che esprimono la capacità di refresh e di reset delle credenziali associate ad un Subject

AUTENTICAZIONE

- LoginModule – interfaccia per l'implementazione delle diverse tecnologie di autenticazione
- LoginContext – offre i metodi per autenticare un Subject e determinare i servizi di autenticazione o i LoginModules configurati per l'applicazione
- Callback – package che offre diverse implementazioni da usare mediante CallbackHandler
- CallbackHandler – interfaccia implementata dall'applicazione e passata al LoginContext, attraverso la quale i LoginModules comunicano bidirezionalmente con gli utenti

AUTORIZZAZIONE

- Policy – classe astratta per definire le politiche di accesso al sistema
- AuthPermission – incapsula i permessi di base richiesti in JAAS per l'autenticazione
- Configuration - rappresenta la configurazione di un LoginModule
- PrivateCredentialPermission – protegge l'accesso alle credenziali di un Subject privato

Identificazione

- Un Subject contiene tre tipologie di informazioni per l'identificazione:
  - Identities - nella forma di uno o più Principals
  - Public credentials - come nome e chiavi pubbliche
  - Private credentials - come password e chiavi private
- Implementazioni dell'interfaccia Principal sono le seguenti classi:
  - **java.security.Identity**
  - **javax.security.auth.kerberos.KerberosPrincipal**
  - **javax.security.auth.x500.X500Principal**
- Qualsiasi Object può rappresentare una credential pubblica o privata

```
Subject subject;
Principal principal;
Object credential;
subject.getPrincipals().add(principal);
subject.getPublicCredentials().add(credential);
subject.getPrivateCredentials().add(credential);
```

Login Modules

- Per costruire un modulo di autenticazione personalizzato occorre implementare l'interfaccia LoginModule
- J2SE 1.4 mette a disposizione un set di LoginModules ready-to-use:
  - **JndiLoginModule** - login con un directory service configurato sotto JNDI (Java Naming and Directory Interface)
  - **Krb5LoginModule** - login con i protocolli di Kerberos v5
  - **NTLoginModule** - login con le informazioni di current user in NT
  - **UnixLoginModule** - login con le informazioni di current user in UNIX
- In **com.sun.security.auth** sono disponibili alcune relative implementazioni dell'interfaccia Principal, come NTDomainPrincipal e UnixPrincipal
- La classe LoginContext carica dinamicamente la configurazione di un modulo di login da un file di testo con la seguente struttura:

```
Application {
   ModuleClass Flag ModuleOptions;
   ModuleClass Flag ModuleOptions;
   .....
};
Application { ..... };
.....
```


Callbacks e CallbackHandlers

- CallbackHandlers e Callbacks raccolgono le informazioni necessarie per l'autenticazione di un utente o un sistema
- Rendono indipendenti i LoginModules dai meccanismi di interazione
- J2SE 1.4 mette a disposizione due CallbackHandlers all'interno del package **com.sun.security.auth.callback**:

  DialogCallbackHandler

  TextCallbackHandler
- Sette Callbacks, presenti in **javax.security.auth.callback**, sono implementate da JAAS e possono essere adoperate per realizzare CallbackHandlers personalizzati:

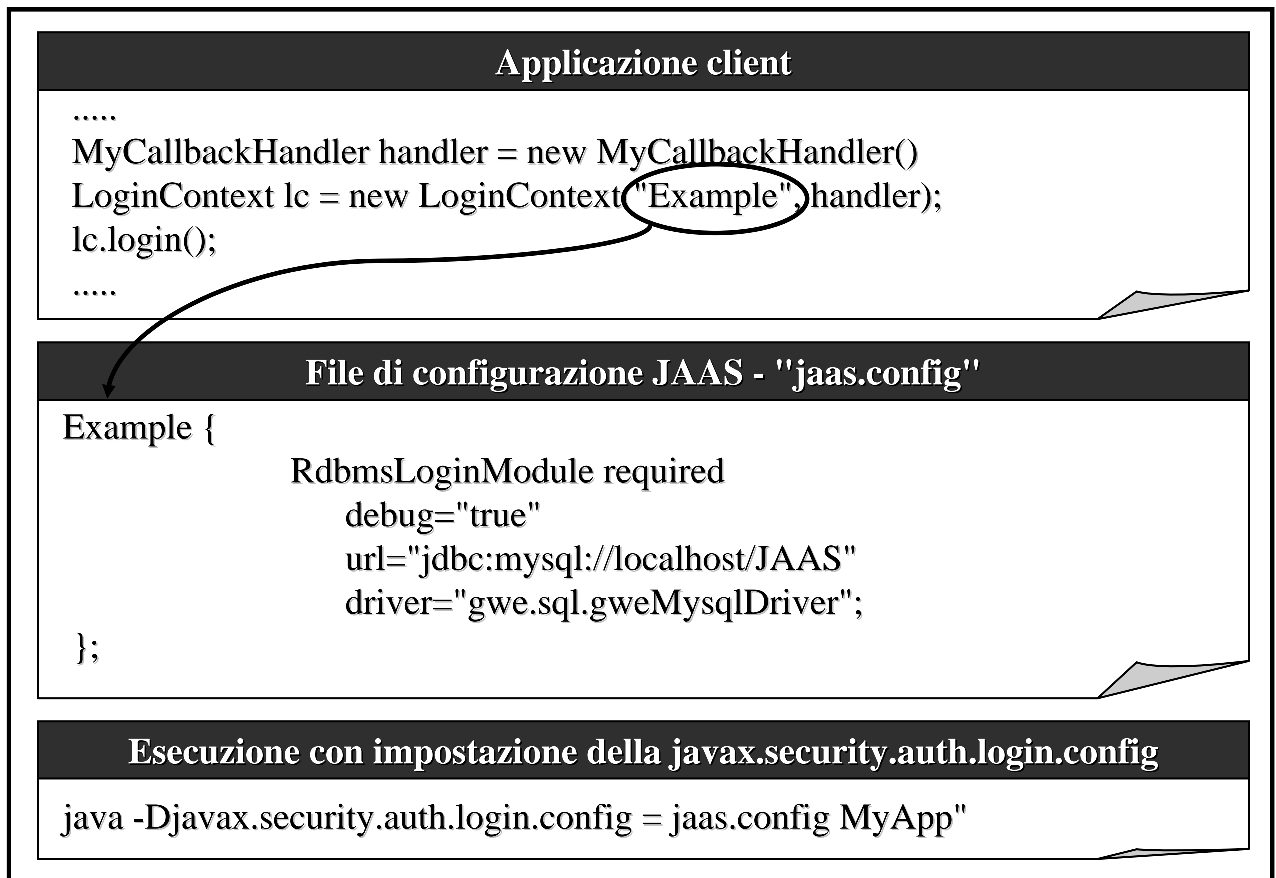
  ChoiceCallback, ConfirmationCallback

  LocaleCallback, NameCallback

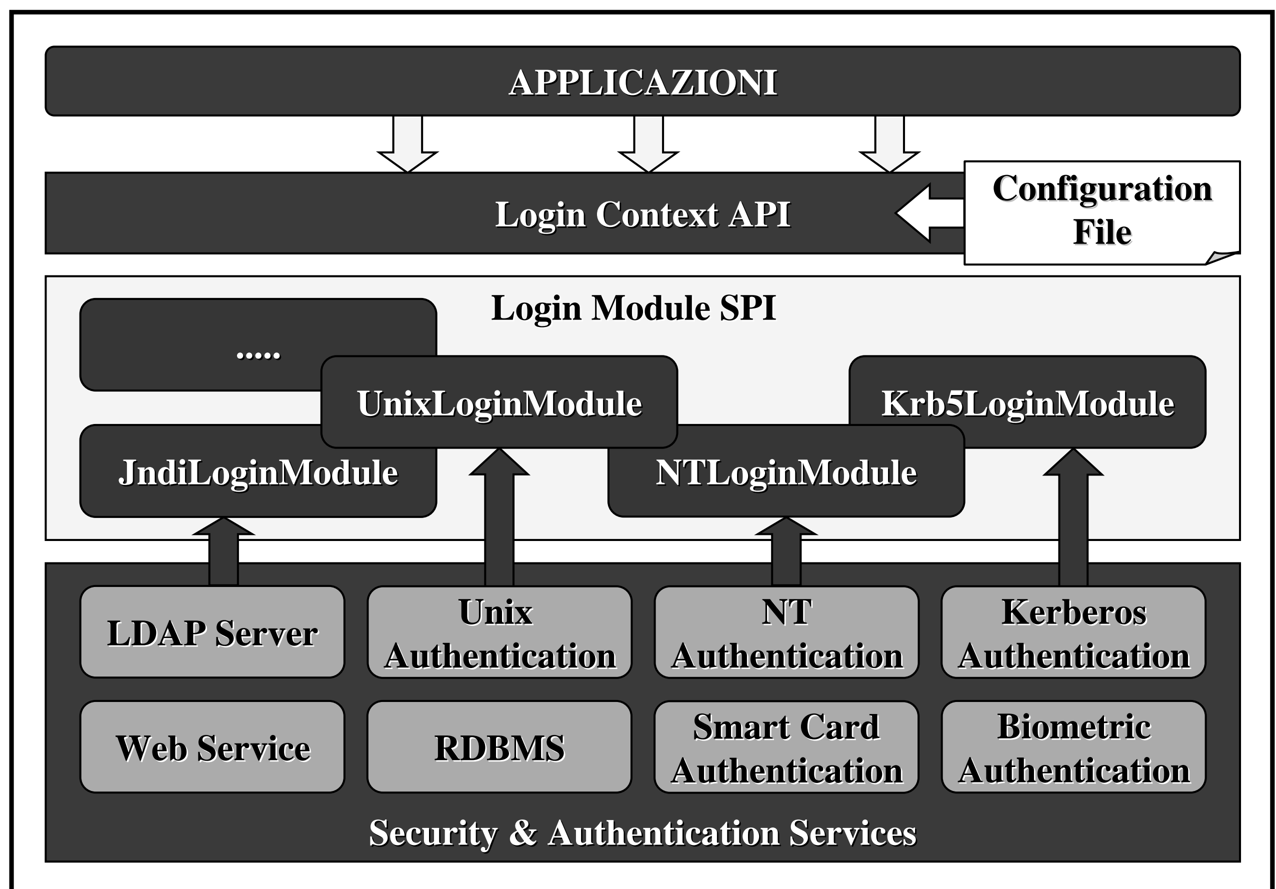
  PasswordCallback, TextInputCallback

  TextOutputCallback

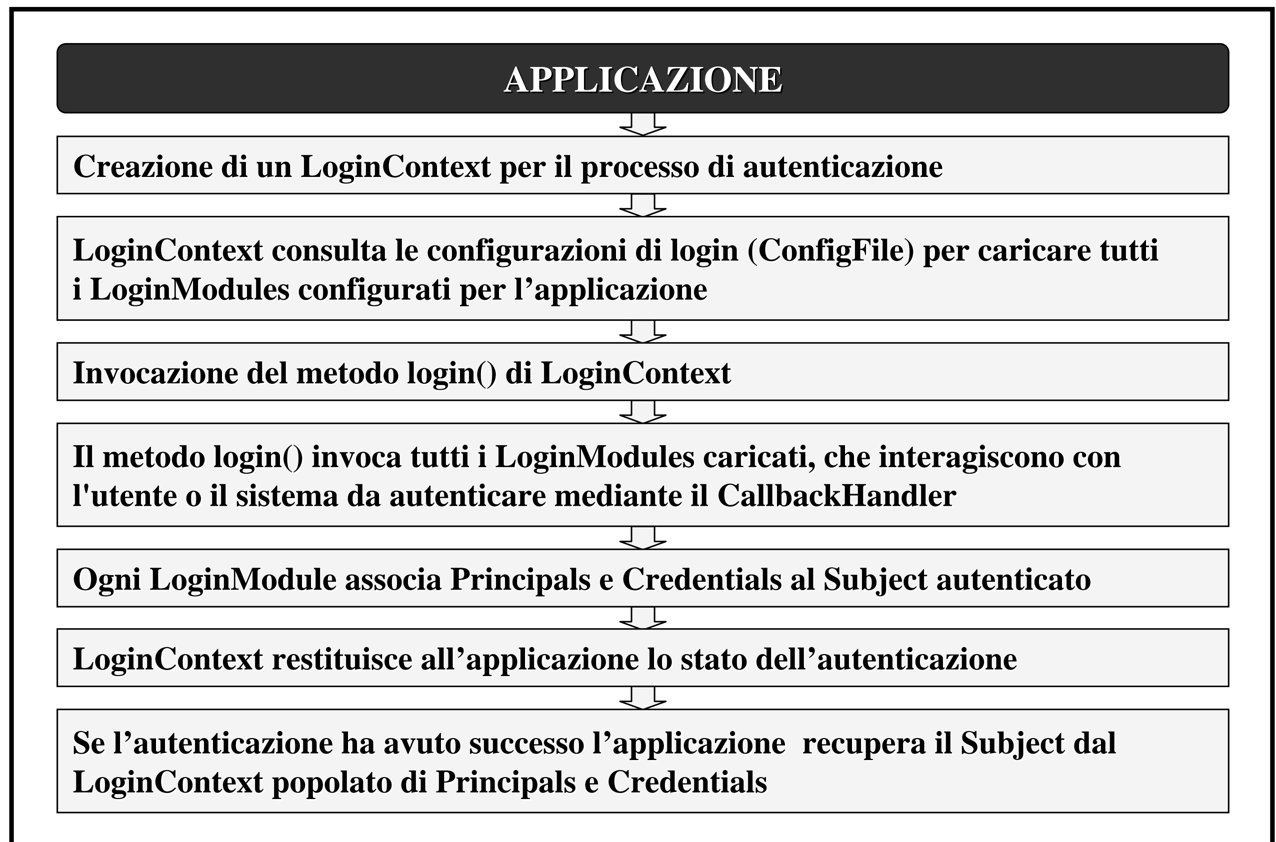
Esempio



Pluggable Authentication



Processo di Autenticazione



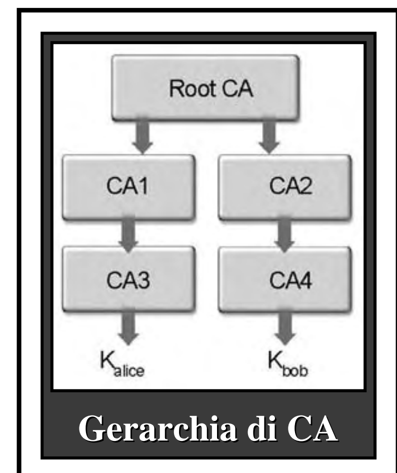
Controllo degli Accessi

- JAAS offre a tempo di esecuzione rinforza il controllo degli accessi ad un sistema
- La classe SecurityManager viene consultata ogni volta che una operazione sensibile debba essere effettuata
- Il SecurityManager a sua volta delega la responsabilità del controllo ad un AccessController
- L'AccessController si procura una immagine dell'AccessControlContext corrente e verifica se si hanno i permessi necessari per l'operazione richiesta
- JAAS offre due metodi che permettono di associare dinamicamente un Subject autenticato con il thread AccessControlContext corrente:
  - **Subject.doAs()**
  - **Subject.doAsPrivileged()**
- Dopo l'associazione ed il controllo dei permessi viene eseguita o meno l'operazione richiesta a seconda delle autorizzazioni concesse

CertPath
Java Certification Path

Caratteristiche

- Java Certification Path (CertPath) offre interfacce ed engine classes per gestire certificati e certification paths, ovvero <u>liste ordinate di certificati</u> derivanti da una gerarchia di Certification Authority (CA)
- Le funzionalità offerte possono essere suddivise in quattro categorie:
  - Basic Certification Path
    - CertPath, CertificateFactory, CertPathParameters
  - Certification Path Validation
    - CertPathValidator, CertPathValidatorResult
  - Certification Path Building
    - CertPathBuilder, CertPathBuilderResult
  - Certificate/CRL Storage
    - CertStore, CertStoreParameters, CertSelector, CRLSelector
- CertPath offre inoltre la possibilità di costruire e validare certification paths di tipo **X.509** secondo gli standards **PKIX**



Gerarchia di CA

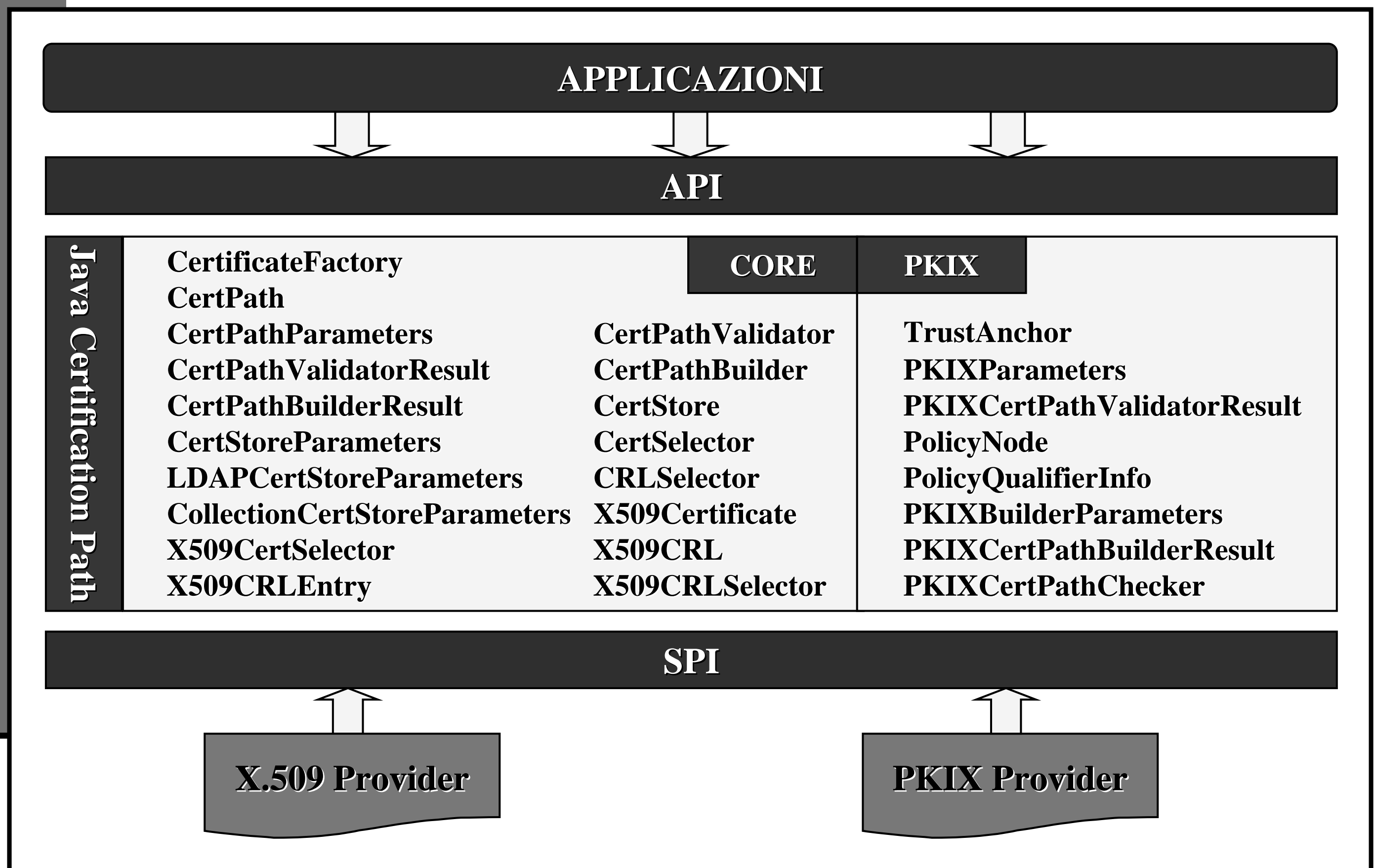
Core Classes

- Le principali CertPath APIs sono contenute in **java.security.cert**:
  - CertPath – definisce le funzionalità comuni per un certification path
  - CertificateFactory – definisce funzionalità di una factory di certificati
  - CertPathParameters – interfaccia per la rappresentazione trasparente del set di parametri usati con un particolare CertPathBuilder o con un algoritmo di validazione
  - CertPathValidator – valida un certification path
  - CertPathValidatorResult – interfaccia per la rappresentazione trasparente del risultato favorevole o dell'output di un algoritmo di validazione di una lista di certicati
  - CertPathBuilder – costruisce un certification path
  - CertPathBuilderResult – interfaccia per la rappresentazione trasparente del risultato o dell'output di un algoritmo di costruzione di un certification path
  - CertStore – fornisce le funzionalità di deposito di una CRL
  - CertStoreParameters – rappresentazione trasparente del set di parametri usati con un particolare CertStore (LDAPCertStoreParameters, CollectionCertStoreParameters)
  - CertSelector / CRLSelector – interfacce per la specifica dell'insieme di criteri adottati per selezionare certificati e CRLs da una collezione di certificati o di CRLs
  - X509Certificate / X509CertSelector – gestiscono certificati X.509
  - X509CRL / X509CRLEntry / X509CRLSelector – gestiscono CRL

PKIX Classes

- Le classi più importanti inerenti gli standard PKIX sono le seguenti:
  - TrustAnchor - rappresenta una "most-trusted" CA usata come root nella validazione di certification paths (non modificabile e thread-safe)
  - PKIXParameters - specifica il set di parametri di ingresso per l'algoritmo PKIX di validazione di certification paths
  - PKIXCertPathValidatorResult - rappresenta il risultato di una validazione
  - PolicyNode - interfaccia che rappresenta un nodo di un albero di policy risultante da un algoritmo di validazione eseguito con successo
  - PolicyQualifierInfo - rappesenta il policy qualifier contenuto nella Certificate Policy Extension del certificato a cui la policy fa riferimento
  - PKIXBuilderParameters - specifica il set di parametri di ingresso per l'algoritmo PKIX di costruzione di certification paths
  - PKIXCertPathBuilderResult - rappresenta il risultato di una costruzione
  - PKIXCertPathChecker - permette di eseguire controlli su certificati X.509

CertPath Architecture



Certificati digitali

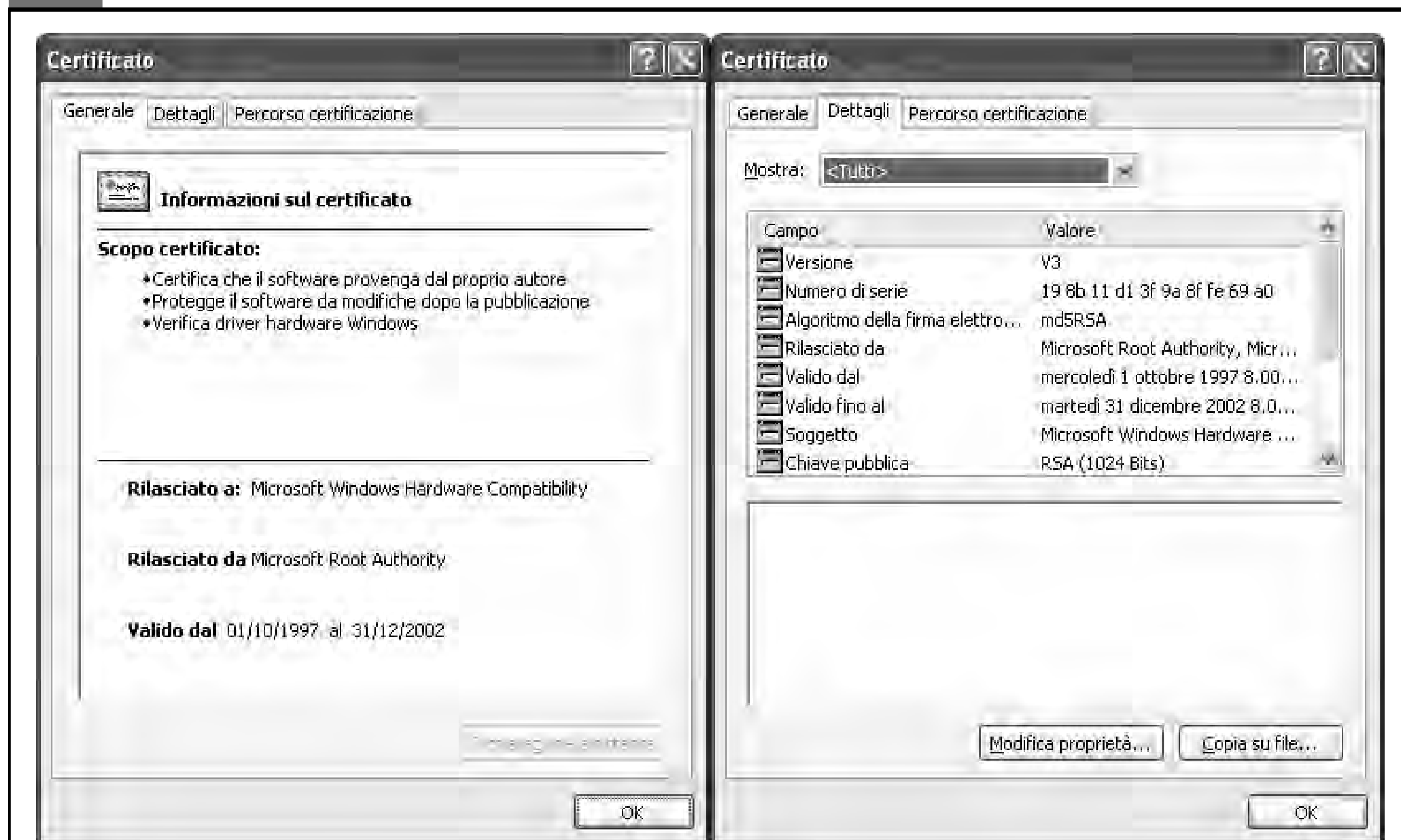
Un certificato digitale è la garanzia fornita da una terza parte che una chiave pubblica appartiene al suo proprietario

- Questa terza parte è detta **Certification Authority** (**CA**)
- Le due CA più note sono Verisign e Thawte (in realtà una filiale di Verisign)
  - Per l'elenco dei certificatori attivi in Italia visitare il sito del Centro Nazionale per l'Informatica nella Pubblica Amministrazione (www.cnipa.gov.it)
- La CA certifica una chiave pubblica firmandola con la sua chiave privata
- Per default, il JDK utilizza i certificati **X.509**:
  - Sono gli standard maggiormente utilizzati per i certificati digitali.
  - Sono definiti in RFC 2459 reperibile all'indirizzo www.ietf.org/rfc/rfc2459.txt.
  - Esistono 3 versioni di X.509: v1,v2 e v3.
  - *v2* e *v3* aggiungono ai certificati alcune informazioni supplementari

Contenuto dei certificati

- Certificato X.509 v1:
  - Versione
    - definisce la versione del certificato: v1,v2,v3
  - Numero seriale del certificato
    - intero univoco che identifica la CA che emette il certificato
  - Identificatore dell'algoritmo di firma
    - definisce l'algoritmo di firma usato dalla CA
  - Periodo di validità
    - definisce il periodo di tempo per cui è valido il certificato.
    - si usano 2 date: *non prima* e *non dopo*
    - il certificato è valido solo nell'intervallo
  - Soggetto
    - indica a chi è stato emesso il certificato
    - i soggetti sono memorizzati con nomi X.500
  - Chiave pubblica del soggetto
  - Firma dell'Autorità di certificazione

Esempio in Windows



Stampa di un certificato DER

```
import java.io.*;
import java.security.cert.*;
.....
String nameCert = “File_Certificate.cer";

FileInputStream fis = new FileInputStream(nameCert);
CertificateFactory cf = CertificateFactory.getInstance("X.509");
X509Certificate x509cert = (X509Certificate)cf.generateCertificate(fis);

System.out.println("\nInformazioni reperite dal certificato: “ +nameCert+"\n");
System.out.println("tipo = " + x509cert.getType());
System.out.println("versione = " + x509cert.getVersion());
System.out.println("soggetto = " + x509cert.getSubjectDN().getName());
System.out.println("inizio validità = " + x509cert.getNotBefore());
System.out.println("fine validità = " + x509cert.getNotAfter());
System.out.println("numero di serie = " + x509cert.getSerialNumber().toString(16));
System.out.println("emettitore = " + x509cert.getIssuerDN().getName());
System.out.println("algoritmo di firma = " + x509cert.getSigAlgName());
System.out.println("algoritmo per la chiave pubblica = " +
                        x509cert.getPublicKey().getAlgorithm());
fis.close();
.....
```

Il certificato deve essere in formato DER (Distinguished Encoding Rules) (estensione .cer)
Un solo certificato per file

Occorre aprire il relativo file ed istanziare un certificato con CertificateFactory inizializzato allo standard X.509
generateCertificate richiede un cast perchè restituisce un Certificate

Certificate Revocation List (CRL)

```
import java.io.*;
import java.security.cert.*;
.....
String nameCRL = “File_CRL.crl";
String nameCert = “File_Certificate.cer";
FileInputStream inCRL = new FileInputStream(nameCRL);
FileInputStream inCert = new FileInputStream(nameCert);

CertificateFactory cf = CertificateFactory.getInstance("X.509");
X509CRL crl = (X509CRL)cf.generateCRL(inCRL);

Certificate certificate = cf.generateCertificate(inCert);

inCRL.close();
inCert.close();

if (crl.isRevoked(certificate)) System.out.println("Il certificato è stato revocato!");
else System.out.println("Il certificato "+nameCert+" e' ok!");
.....
```

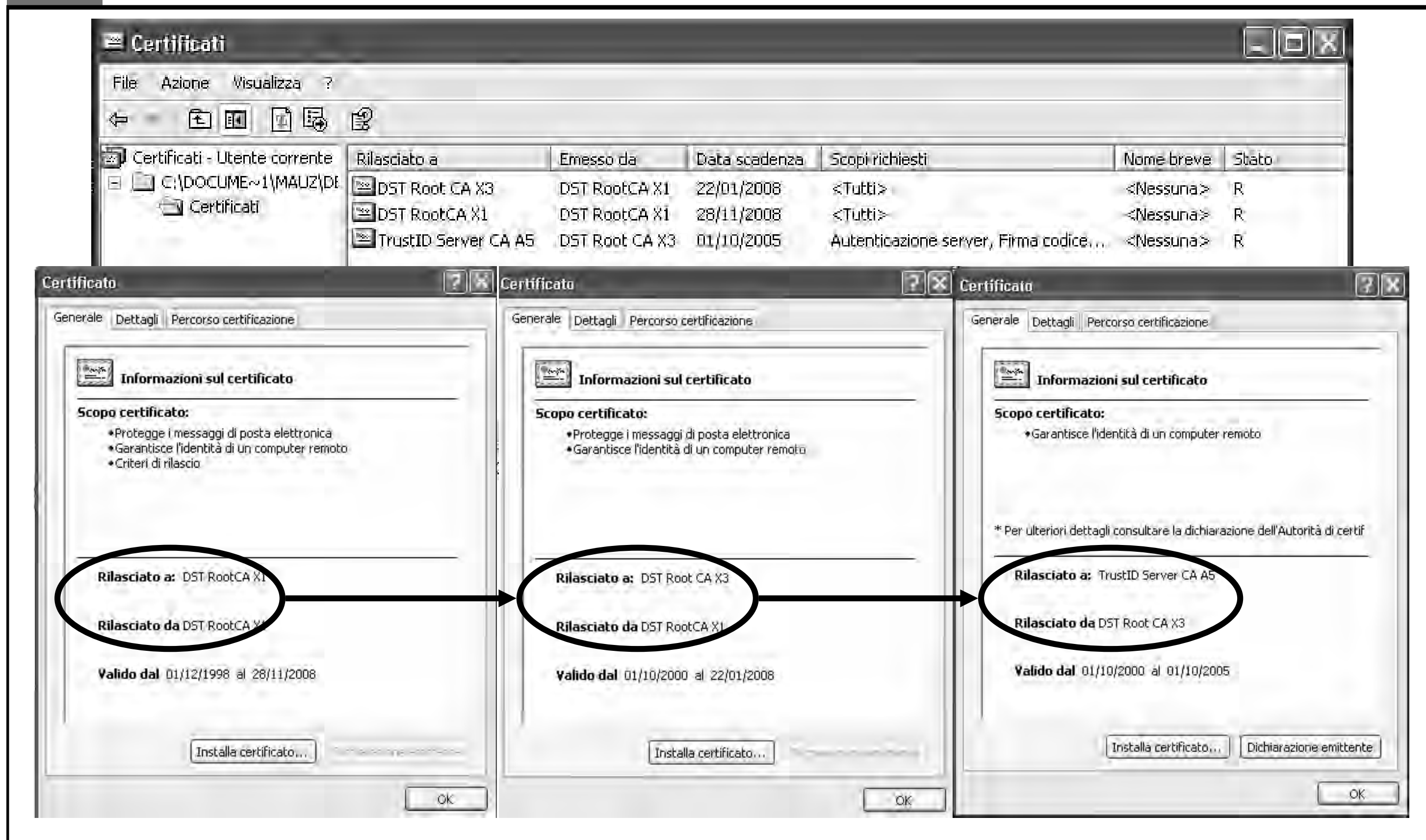
Il file deve contenere una CRL (estensione .crl)

Creazione di una CRL di X.509

Verifica del cetificato attraverso il metodo isRevoked()

Esempio di Certification Path

- CertPath Root della Digital Signature Trust (DST Root Cert.p7b):



Stampa di un certification path

```
import java.io.*;
import java.security.cert.*;
import java.util.Collection;
import java.util.Iterator;
.....
String nameCert = “File_Certificates.p7b";

FileInputStream fis = new FileInputStream(nameCert);
CertificateFactory cf = CertificateFactory.getInstance("X.509");
Collection c = cf.generateCertificates(fis);

Iterator it = c.iterator();
int j = 1;
while (it.hasNext()) {
    System.out.println("certificato N°."+j+ ": ");
    X509Certificate x509cert = (X509Certificate)it.next();
    System.out.println(x509cert);
    j++;
}
fis.close();
.....
```

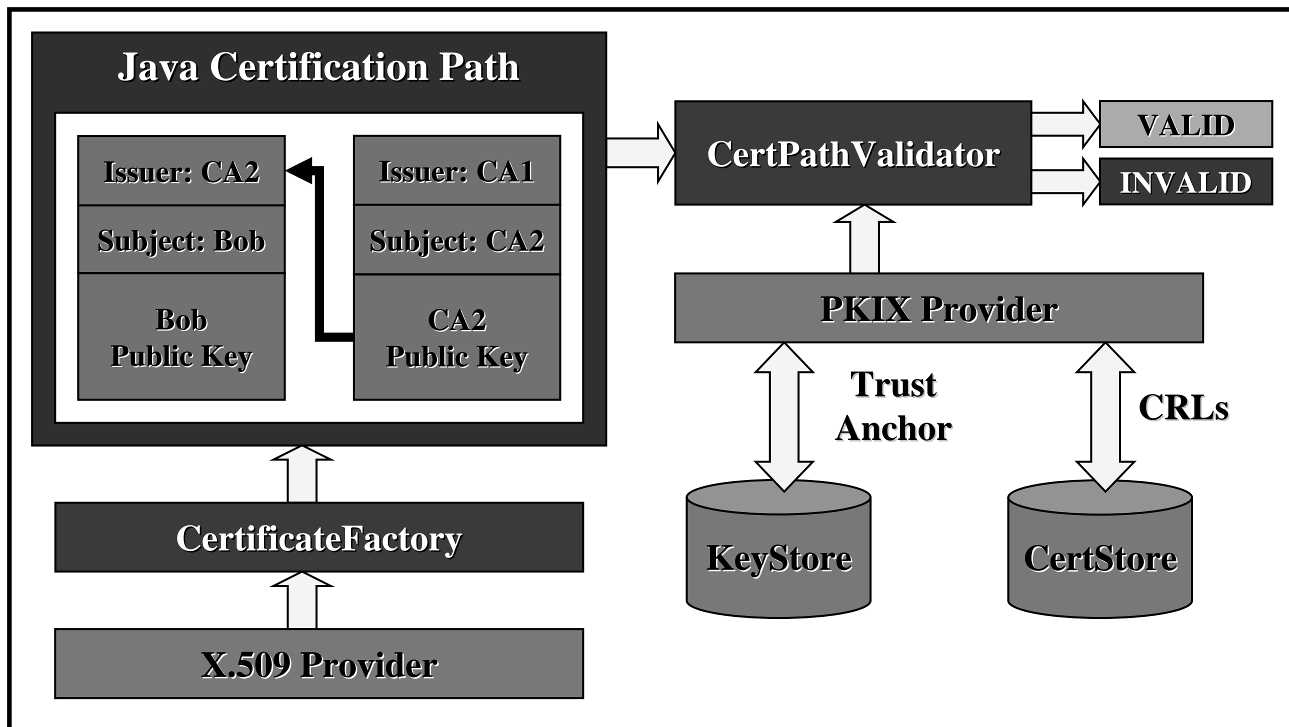
Il file deve contenere una catena di certificati in formato PKCS#7 (estensione .p7b) o una sequenza di certificati di formato DER (estensione .cer)

Iterator per navigare all'interno della collezione di certificati

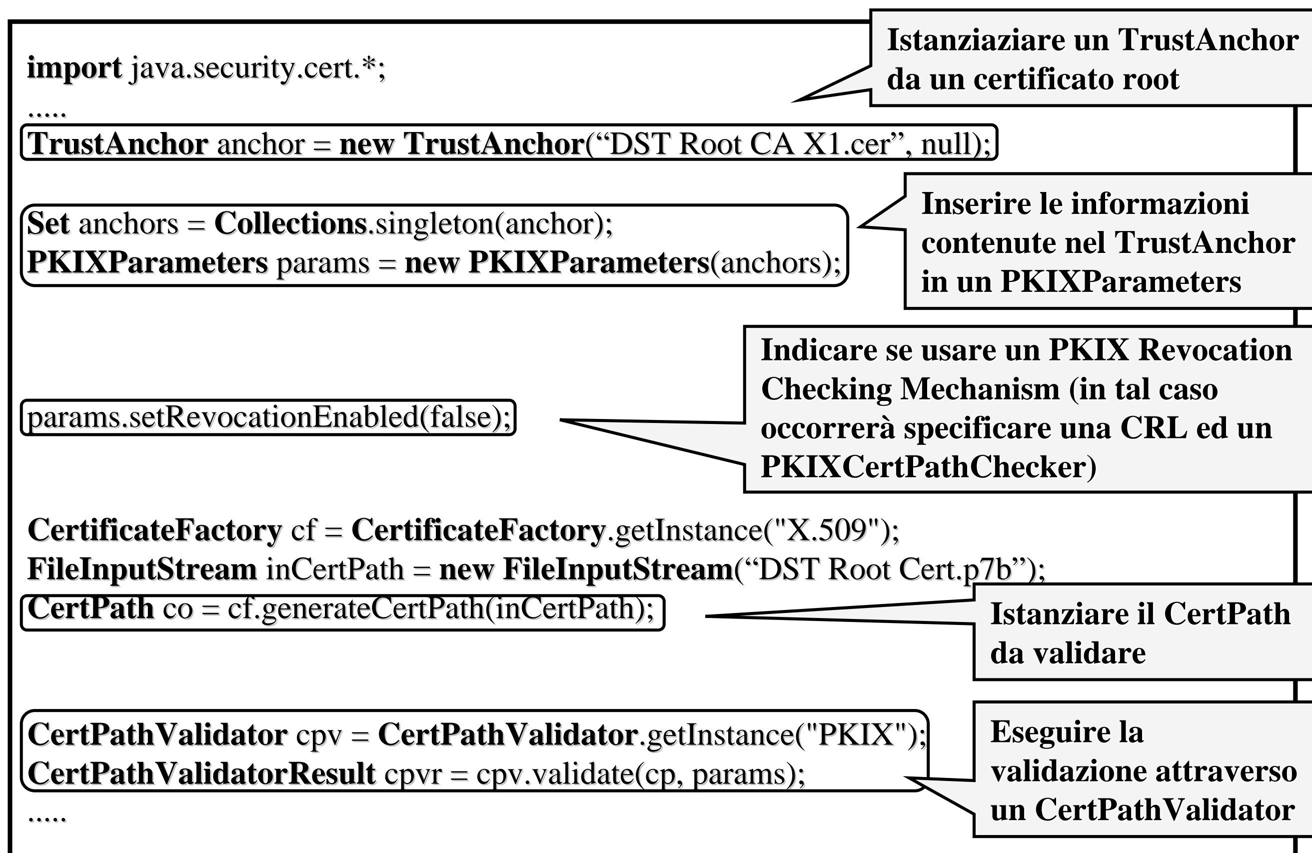
Tutto funziona correttamente solo se l'InputStream utilizzato supporta i metodi mark() e reset()
In caso contrario generateCertificates(),che si avvale di tali metodi, consumerà tutto lo stream di ingresso in un colpo solo
I vari certificati vengono inseriti in una Collection

Stampa a video usando la toString() di X509Certificate

Schema di Validazione



Processo di validazione



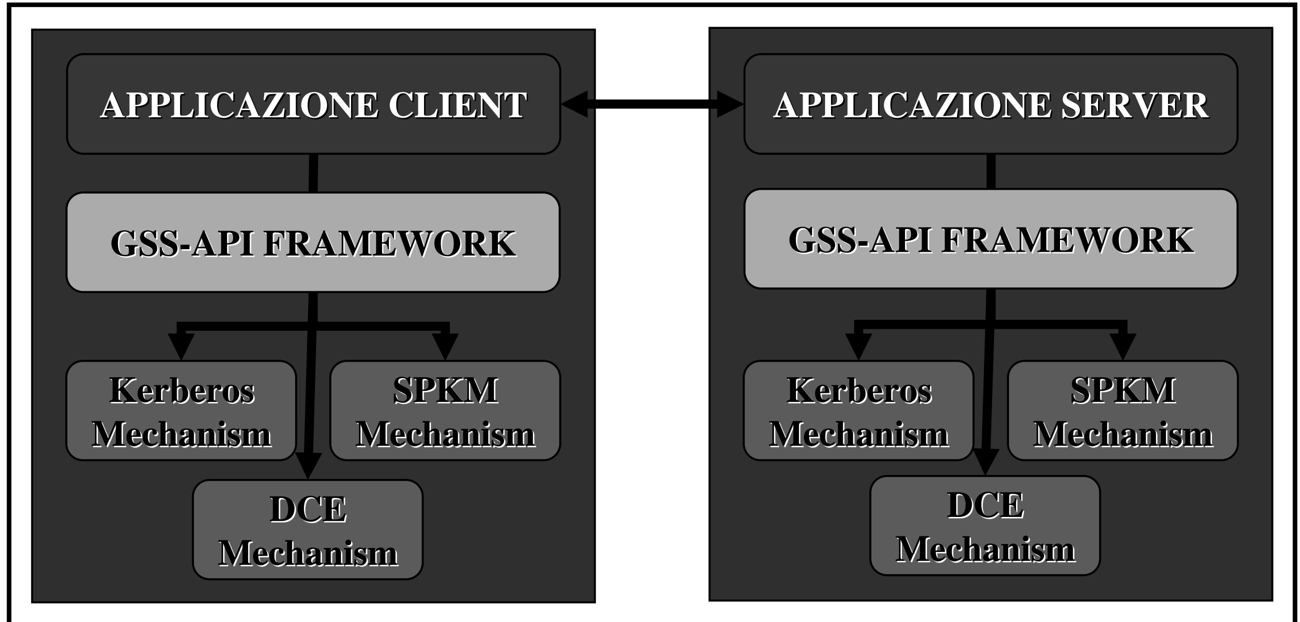
JGSS
Java Generic Security Service

Caratteristiche

- Java Generic Security Service (JGSS) contiene le Java bindings per la Generic Security Services Application Program Interface (GSS-API)
- Le GSS-API sono state progettate dal Common Authentication Technology working group dell'IETF per offrire un accesso uniforme a servizi di autenticazione e comunicazione peer-to-peer che adottano meccanismi di sicurezza sottostanti, isolando il chiamante dai dettagli implementativi
- Tali meccanismi di sicurezza possono essere impiegati in maniera simultanea e selezionati dall'applicazione a tempo di esecuzione
- Le GSS-API offrono i seguenti servizi di sicurezza:
  - autenticazione
  - riservatezza ed integrità dei messaggi
  - sequencing dei messaggi protetti
  - replay detection
  - credential delegation

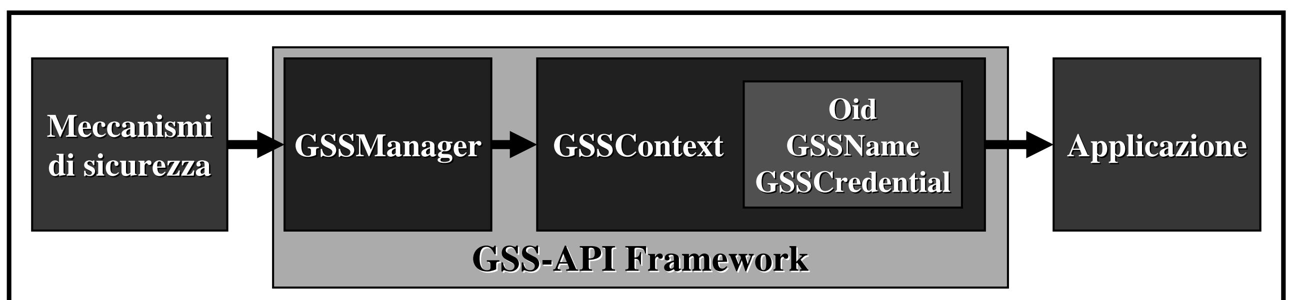
Meccanismi di sicurezza

- IETF ha definito i seguenti due meccanismi di sicurezza:
  - Kerberos v5
  - Simple Public Key Mechanism (SPKM)
- Altra tecnologia utilizzata è DCE (Distributed Computing Environment)



Kerberos v5 in JGSS

- JGSS offre la possibilità di implementare il servizio Kerberos v5 con l'ausilio dei meccanismi offerti da JAAS
- Classi principali in JGSS:
  - la classe GSSManager conosce i componenti e i meccanismi di sicurezza sottostanti ed è responsabile della loro invocazione a run-time
  - la classe Oid rappresenta l'Universal Object Identifier
  - l'interfaccia GSSName rappresenta una entità generica da instanziare e permette di definire l'user duke del servizio sicuro implementato
  - l'interfaccia GSSCredential incapsula le credentials possedute da un'entità
  - l'interfaccia GSSContext permette di definire i servizi di sicurezza offerti alle applicazioni comunicanti



Kerberos v5 in JAAS

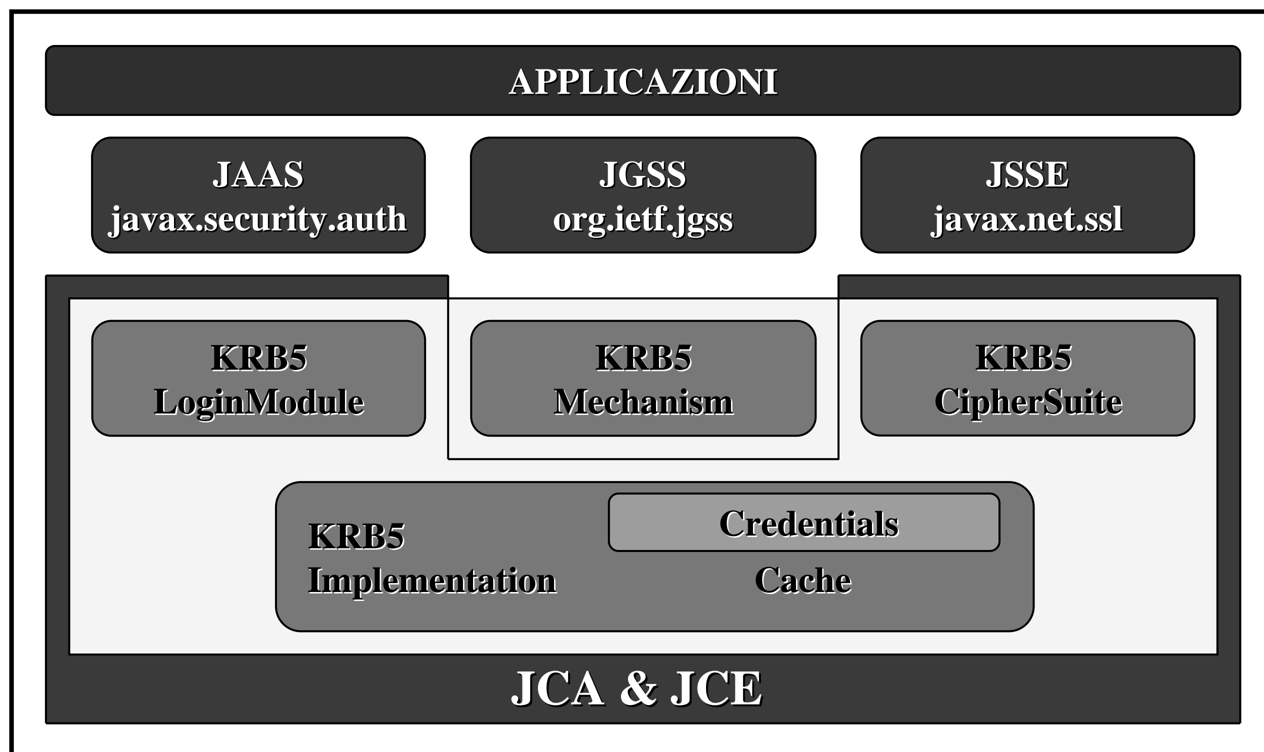
- JAAS mette a disposizione degli sviluppatori il modulo di autenticazione Krb5LoginModule e le classi del package **javax.security.auth.kerberos**:
  - DelegationPermission - usata per limitare l'uso del Kerberos delegation model (forwardable and proxiable tickets)
  - KerberosKey - incapsula un segreto a lungo termine per un Kerberos principal
  - KerberosPrincipal - incapsula un Kerberos principal
  - KerberosTicket - incapsula un Kerberos ticket e le informazioni ad esso associate dal punto di vista del cliente. Racchiude tutte le informazioni che il Key Distribution Center (KDC) invia al client nel messaggio di risposta KDC-REP definito in Kerberos Protocol Specification (RFC 1510)
  - ServicePermission - usata per proteggere i servizi Kerberos e le credenziali necessarie per accedere a questi servizi. Contiene un service principal name e una lista di operazioni che specificano il contesto in cui le credenziali possono essere usate

JGSS vs JSSE

- JGSS e JSSE presentano alcune analogie in termini di strumenti per comunicazioni sicure peer-to-peer
- Sono accomunati dalle seguenti caratteristiche di sicurezza:
  - autenticazione client/server
  - protezione della riservatezza e dell'integrità dei dati trasmessi
- Si differenziano per i seguenti aspetti:
  - Kerberos Single Sign-On Support
  - Communications API
  - Credential Delegation
  - Selective Encryption
  - Protocol Requirements

(When to use Java GSS-API vs. JSSE)

Kerberos v5 in J2SE



Autenticazione

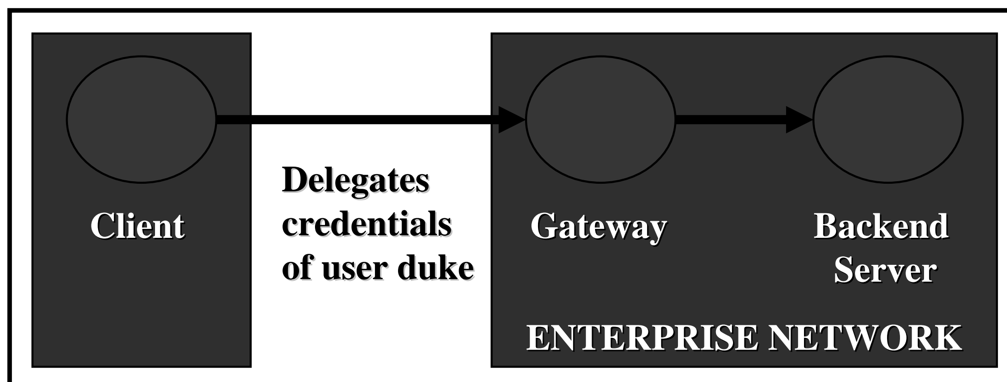
- Prima che un GSSContext possa essere usato per i servizi di sicurezza offerti, occorre realizzare uno scambio di token per l'autenticazione tra le applicazioni comunicanti
- GSSContext offre i metodi necessari a tale scambio:
  - **initSecContext()** - invia il token all'applicazione paritaria
  - **acceptSecContext()** - riceve il token
- Autenticazione in Kerberos V5:
  - Il client per primo invia il token costruito con **initSecContext()** contenente il messaggio AP-REQ di Kerberos
  - Il provider Kerberos ottiene dal TGT (Kerberos Ticket) del client un ticket di servizio per il target server, che viene cifrato ed inserito nel messaggio
  - Il server riceve il token e lo decifra con il metodo **acceptSecContext()** che autentica il client generando un nuovo token
  - In caso di autenticazione mutua, il nuovo token viene adoperato per l'autenticazione del server e rispedito al client con **initSecContext()**

Protezione dei messaggi

- Il contesto sicuro stabilito (GSSContext) offre protezione delle comunicazioni mediante riservatezza ed integrità dei messaggi
- I metodi relativi sono:
  - **wrap()** - permette di incapsulare il testo in chiaro o cifrato in un token che ne protegge l'integrità, da inviare all'applicazione paritaria
  - **unwrap()** - rimette in chiaro il testo originario dal token ricevuto
- L'oggetto trasmesso contiene informazioni sul testo originario (se in chiaro o cifrato) e relativi warnings per il controllo del sequencing e della duplicazione dei messaggi

Credential Delegation

- Java GSS-API aiuta il client a comunicare in maniera sicura le proprie credenziali al server e realizzare un nuovo contesto sicuro
- Nel caso di Kerberos V5, le credenziali delegate sono rappresentate da un forwarded TGT incapsulato nel primo token spedito dal client al server
- Usando questo TGT il server può ottenere un ticket service legato al client per alcuni altri servizi



Default Credential Acquisition Model

Invocazione del metodo di login del modulo JAAS Kbr5LoginModule

↓

Krb5LoginModule ottiene un TGT per il client dal KDC o da una cache preesistente e lo memorizza nel private credentials set del Subject. Sul lato server Kbr5LoginModule memorizza nel Subject, accanto al KerberosTicket, una KerberosKey per il server, adoperata nei passi 5-6-7 per decifrare il ticket service che il client invia

↓

Il client riceve il Subject popolato e invoca il metodo Subject.doAs() o il metodo Subject.doAsPrivileged() per inserire il Subject nel contesto di controllo d'accesso del thread ClientAction in esecuzione

↓

ClientAction invoca il metodo GSSContext.createCredential() passando il Kerberos V5 OID (unique object identifier) in desiredMechs

↓

GSSContext.createCredential() invoca il provider di Kerberos V5 richiedendo una Kerberos credential per iniziare un contesto sicuro

↓

Il provider ottiene il Subject dal corrente contesto di controllo degli accessi e ricerca il suo private credential set per un valido KerberosTicket (TGT) per l'utente

↓

Il KerberosTicket viene fornito al GSSManager che lo memorizza in un GSSCredential da restituire al chiamante

PKCS
Public Key Cryptography Standards

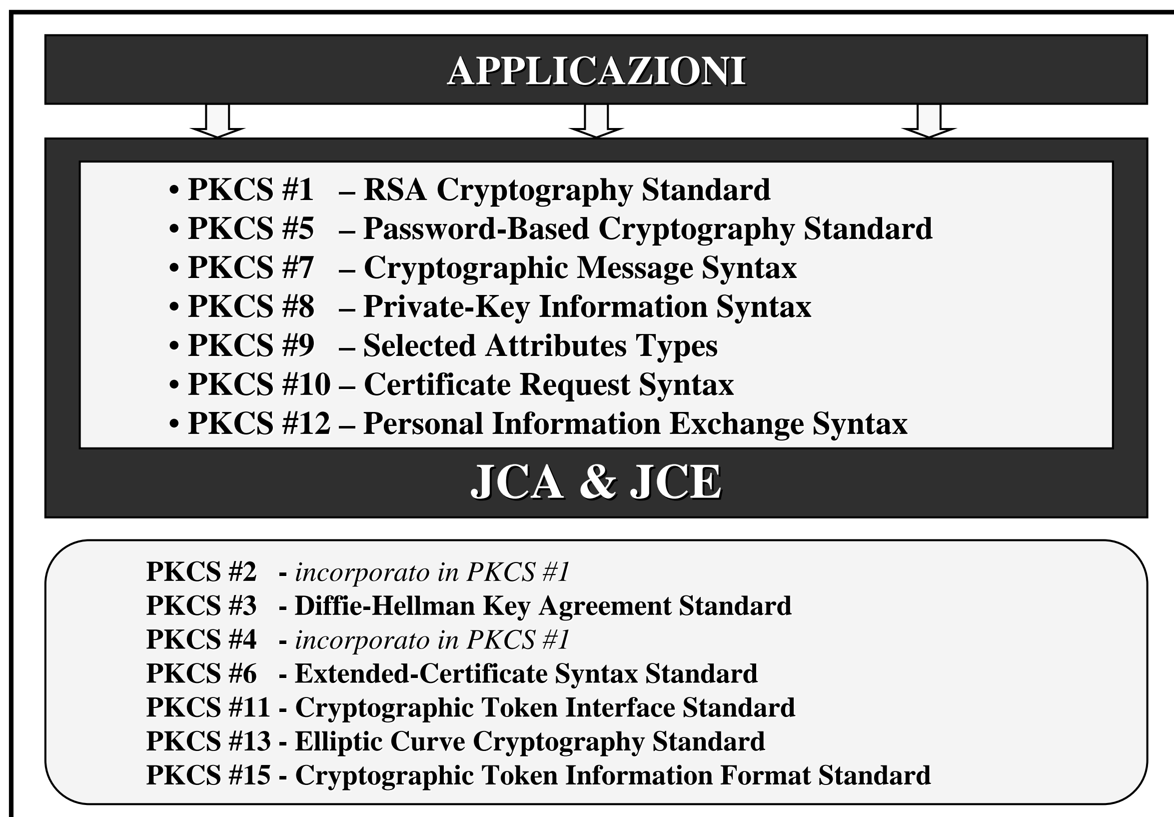
Public Key Cryptography Standards

- Le specifiche PKCS sono state prodotte dagli RSA Laboratories con lo scopo di accelerare lo sviluppo della crittografia a chiave pubblica
- Pubblicate nel 1991, sono state largamente implementate ed integrate in molti standard de facto, come **ANSI X9**, **PKIX**, **SET**, **S/MIME** e **SSL**
- JCA e JCE implementano gli standards **PKCS #1**, **#5**, **#7**, **#8**, **#9**, **#10** e **#12**
- J2SDK 1.4 Beta 3 include un miglior supporto per gli standard PKCS 1.0 con un set di APIs contenuto in JSR (Java Specification Request) 74:

  javax.security.pkcs.pkcs1
  javax.security.pkcs.pkcs5
  javax.security.pkcs.pkcs7
  javax.security.pkcs.pkcs8
  javax.security.pkcs.pkcs9
  javax.security.pkcs.pkcs10
  javax.security.pkcs.pkcs12

- La versione PKCS 2.1 presenta diversi miglioramenti che saranno implementati ed aggiunti in JSR 74

PKCS Architecture



Java Keytool Key and Certificate Management Tool

Key Pair Generating

- Generazione di una coppia di chiavi asimmetriche e inserimento dei dati identificativi dell'utente:

keytool -genkey -keystore keystore -keyalg rsa –keysize 1024 -alias mauz -validity 180

Immettere la password del keystore: 20Colleluori03Maurizio1980
Specificare nome e cognome [Unknown]: Maurizio Colleluori
Specificare il nome dell'unità aziendale [Unknown]: MM
Specificare il nome dell'azienda [Unknown]: Mauz
Specificare la località [Unknown]: Bologna
Specificare la provincia [Unknown]: Bologna
Specificare il codice a due lettere del apese in cui si trova l'unità [Unknown]: IT
Il dato CN=Maurizio Colleluori, MM, O=Mauz, L=Bologna, ST=Bologna, C=IT è corretto? [no]: si
Immettere la password della chiave per <mauz>
(INVIO se corrisponde alla password del keystore): 20031980

Le chiavi private hanno associato un certificato che autentica le corrispondenti chiavi pubbliche. L'accesso ad un keystore è protetto da una password definita al momento della sua creazione. Si può decidere di proteggere la chiave privata con una password differente

Certificate Signing Request

- Produzione della richiesta di certificato (Certificate Signing Request) auto-firma del certificato provvisorio con la chiave privata:

  keytool -certreq -keystore keystore -file csr.txt -alias mauz

Immettere la password del keystore: 20Colleluori03Maurizio1980
Immettere la password della chiave per <mauz>: 20031980

Richiesta per ogni accesso al keystore!

Richiesta per ogni accesso alla chiave segreta!

File csr.txt

-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBqzCCARQCAQAwazELMAkGA1UEBhMCSVQxEDAOBgNVBAgT[illegible]VBAcTB0Jv
bG9nbmExDTALBgNVBAoTBE1hdXoxCzAJBgNVBAsTAk1NMRwwGg[illegible]pbyBDb2xs
ZWx1b3JpMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQD0qUrx5YEOwgHlufHMmB1QEJ4FQj4d
0kY4JVPzXNBTByYis9QXLDyKL9nFso8/qMxMesu0UAEKZJsLvNX8vOMv7dcudsfuLzWXqrbvyiC1
Q5wxtXdYi3Rx5l7txAbbbVcpj9BMX6lLOz7wpxcmJzv/Y7gVlo5aclnfpOQopzxRewIDAQABoAAw
DQYJKoZIhvcNAQEEBQADgYEA1hDqkWAttZEzNHAEncUodsUr0Ak8pfKhhw+BL4gmOkPo78jdqCaP
o1QIsX6KOB0X9s/+2MbWH3nVlVZP4HyQ8bv8xbDs6cY3m+adN7I01+7UHIiWqIUV2WKKmBGZSaQ+
gPNM/oV7dMZcH8mXr9zQObPqcHKqBDNjqtQC0IhE21o=
-----END NEW CERTIFICATE REQUEST-----

Self Certificate

- Generare un certificato X.509 auto-firmato:

  keytool -selfcert -keystore keystore -keyalg rsa –keysize 1024 -alias mauz -validity 180

Generale | Dettagli | Percorso certificazione

Informazioni sul certificato

Questo certificato di origine CA non è considerato attendibile. Per renderlo attendibile, installarlo nell'archivio dell'Autorità di certificazione fonti attendibili.

Rilasciato a: Maurizio Colleluori

Rilasciato da Maurizio Colleluori

Valido dal 24/07/2003 al 20/01/2004

Installa certificato...

Mostra: <Tutti>

Campo	Valore
Valido dal	giovedì 24 luglio 2003 19.09.04
Valido fino al	martedì 20 gennaio 2004 19.0...
Soggetto	Maurizio Colleluori, MM, Mauz, ...
Chiave pubblica	RSA (1024 Bits)
Algoritmo di identificazione ...	sha1
Identificazione personale	2c 8e 41 97 44 25 26 b3 1d 01...

```
30 81 89 02 81 81 00 f4 a9 4a f1 e5 81 0e
c2 01 e5 b9 f1 cc 98 1d 50 10 9e 05 42 3e
1d d2 46 38 25 53 f3 5c d0 53 07 26 22 b3
d4 17 2c 3c 8a 2f d9 c5 b2 8f 3f a8 cc 4c
7a cb b4 50 01 0a 64 9b 0b bc d5 fc bc e3
2f ed d7 2e 76 c7 ee 2f 35 97 aa b6 ef ca
20 b5 43 9c 31 b5 77 58 8b 74 71 e6 5e ed
c4 06 db 6d 57 29 8f d0 4c 5f a9 4b 3b 3e
f0 a7 17 26 27 3b ff 63 b8 15 96 8e 5a 72
```

Copia su file...

Altre funzionalità

- Download del certificato firmato dalla CA (estensione .cer secondo il formato PKCS7) e aggiornamento del certificato provvisorio auto-fimato:

  keytool -import -keystore keystore -file file.cer -alias mauz

- Esportazione di un certificato in un file esterno:

  keytool -import -keystore keystore -file file.cer -alias mauz

- Visualizzazione di una entry del keystore:

  keytool -list -keystore keystore -alias mauz

- Lettura di un file.cer:

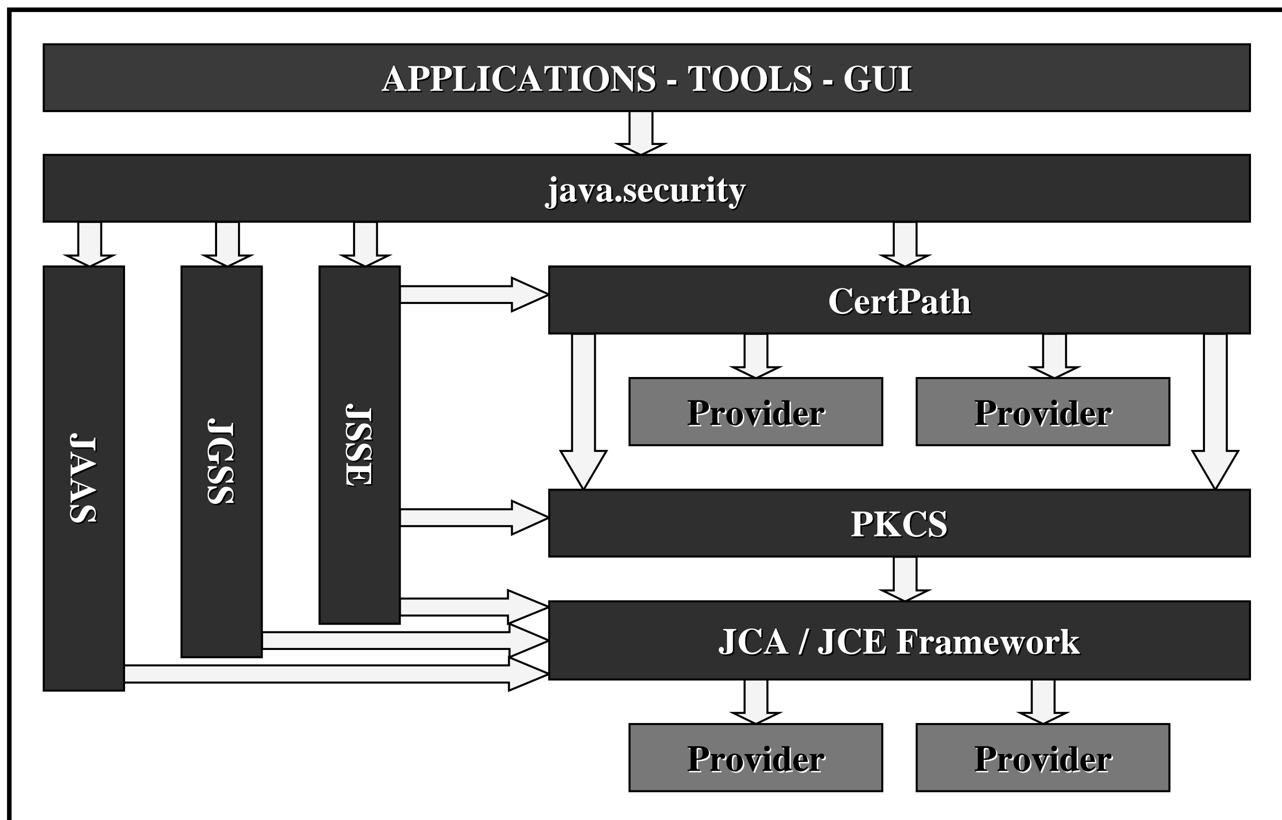
  keytool -printcert -file file.cer

- Altre utilità:

keytool -keyclone ...	clonazione di una entry del keystore
keytool -storepasswd ...	modifica della password del keystore
keytool -keypasswd ...	modifica della password della chiave privata
keytool -delete ...	cancellazione di una entry del keystore

Java Security Roadmap

Java Security Roadmap



Java Security Enhancements in J2SE5

Security Enhancements in J2SE5

- **JCE**
  - Diversi nuovi algoritmi sono stati introdotti nel provider SunJCE
  - Nuove API per facilitare il supporto a ECC
  - Supporto alla cifratura con RSA

  (Per dettagli: JCE Enhancements)
- **JSSE**
  - SSLEngine in grado di gestire implementazioni non-blocking SSL/TLS
  - Supporto per provider SSL/TLS pluggable prodotti da terze parti, ma con restrizioni sugli algoritmi di cifratura
  - Supporto per i cifrari legati a Kerberos
  - Impostazione di un CertPath-based TrustManager come default JSSE TrustManager
  - Possibilità di utilizzare JCE per la cifratura in modo esclusivo
  - Supporto per SSL/TLS-based RMI Socket Factories

  (Per dettagli: JSSE Reference Guide e SSL/TLS-based RMI Socket Factory classes)
- **JGSS**
  - Nuove implementazioni Java per Kerberos, inclusi un rinnovato TGT ed un nuovo supporto per la cifratura con Triple DES

  (Per dettagli: JGSS/Kerberos Enhancements)

Security Enhancements in J2SE5.

- E ancora:
  - Supporto per il Time-Stamp Protocol (TSP) definito in RFC 3161

  (Per dettagli: Signature Timestamp Support)
  - Supporto per il Simple Authentication and Security Layer, o SASL, e i diversi meccanismi SASL attraverso le Java SASL API

  (Per dettagli: Java SASL reference guide)
  - Supporto per PKCS#11 come provider JCE per garantire una migliore accelerazione hardware e un migliore interfacciamento con le Smartcards

  (Per dettagli: PKCS#11 Guide)
  - Supporto per l'On-Line Certificate Status Protocol (OCSP) definito in RFC 2560
  - Migliore conformità per le CertPath API con PKIX (RFC 3280)
  - Implementazione di un PKCS#12

  (Per dettagli: PKI Enhancements)

Riferimenti

J2SDK v1.4.2 Security Documentation

- http://java.sun.com/j2se/1.4.2/docs/guide/security/index.html
  - Security enhancements for the JavaTM 2 SDK, Standard Edition, v 1.4.2
  - Security enhancements for the JavaTM 2 SDK, Standard Edition, v 1.4.1
  - Security enhancements for the previous release, JavaTM 2 SDK, Standard Edition, v 1.4
  - Security Guides - Security API Specification (javadoc)
    - General Security
    - Certification Path
    - JAAS
    - Java GSS-API
    - JCE
    - JSSE
  - Security Tools
  - Security Tutorials
  - For More Information

Java 2 Standard Edition 5.0

- New Features and Enhancements J2SE 5.0
- J2SE 5.0 in a Nutshell
- Java 2 SDK v1.5.0 Security Documentation
- Security Enhancemenents in the Java 2 SDK v1.5.0

Standards

- Public Key Cryptography Standards (PKCS)
  http://www.rsasecurity.com/rsalabs/pkcs/
- Java Specification Request (JSR) 74
  http://www.jcp.org/en/jsr/detail?id=74

Conferenze JavaOne

- http://java.sun.com/javaone/index.jsp
  - JavaOne 2002 Presentations (PDF documents):
    - Java 2 Platform, Standard Edition (J2SE) Security: Present and Future
    - Authentication and Single Sign-On
    - Networking with Java 2 Platform, Standard Edition (J2SE): Present and Future
    - Single Sign-on Using the JAAS and Java GSS APIs
    - Securing the Wire
    - Java Naming and Directory Interface (JNDI)
    - JSR 105: XML Digital Signature API
  - JavaOne 2001 Presentations (PDF documents):
    - Java 2 Platform, Standard Edition (J2SE) Security: Present and Future
    - Security for the J2SE: CertPath, JCE, and JSSE
    - Java 2 Platform, Standard Edition (J2SE) Networking: Present and Future
    - Java 2 Platform, Standard Edition (J2SE) Networking and IPv6
    - Unlocking Public Key Technologies for the Java Platform Update: JSR74
    - Security for the J2SE Platform: JAAS, Java GSS-API & Kerberos
    - Security for the Java 2 Platform, Standard Edition (J2SE) and XML

Articoli e FAQ

- FAQs, Whitepapers, Articles
  - Applet Security FAQ
  - Single Sign-on Using Kerberos in Java
  - JAAS white paper
  - Secure Computing With Java: Now and the Future

Alcune applicazioni

- JSSE
  - RMI – SOCKETS – URLS
    http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/samples/index.html
  - SSL CLIENT / SERVER
    http://www.javaworld.com/javaworld/jw-05-2001/jw-0511-howto.html
  - HTTPS
    http://www.javaworld.com/javaworld/javatips/jw-javatip111.html
  - SECURE MAIL
    http://www.javaworld.com/javaworld/javatips/jw-javatip115.html
- JAAS
  - http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-jaas.html
- Keytool
  - http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html