Corso di Laurea Specialistica in Ingegneria Informatica
Facoltà di Ingegneria - A.A. 2003/04
Università degli studi di Bologna

Java Security Extensions

Corso di Tecnologie per la Sicurezza LS
Prof. R. Laschi

Maurizio Colleluori
e-mail: 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:

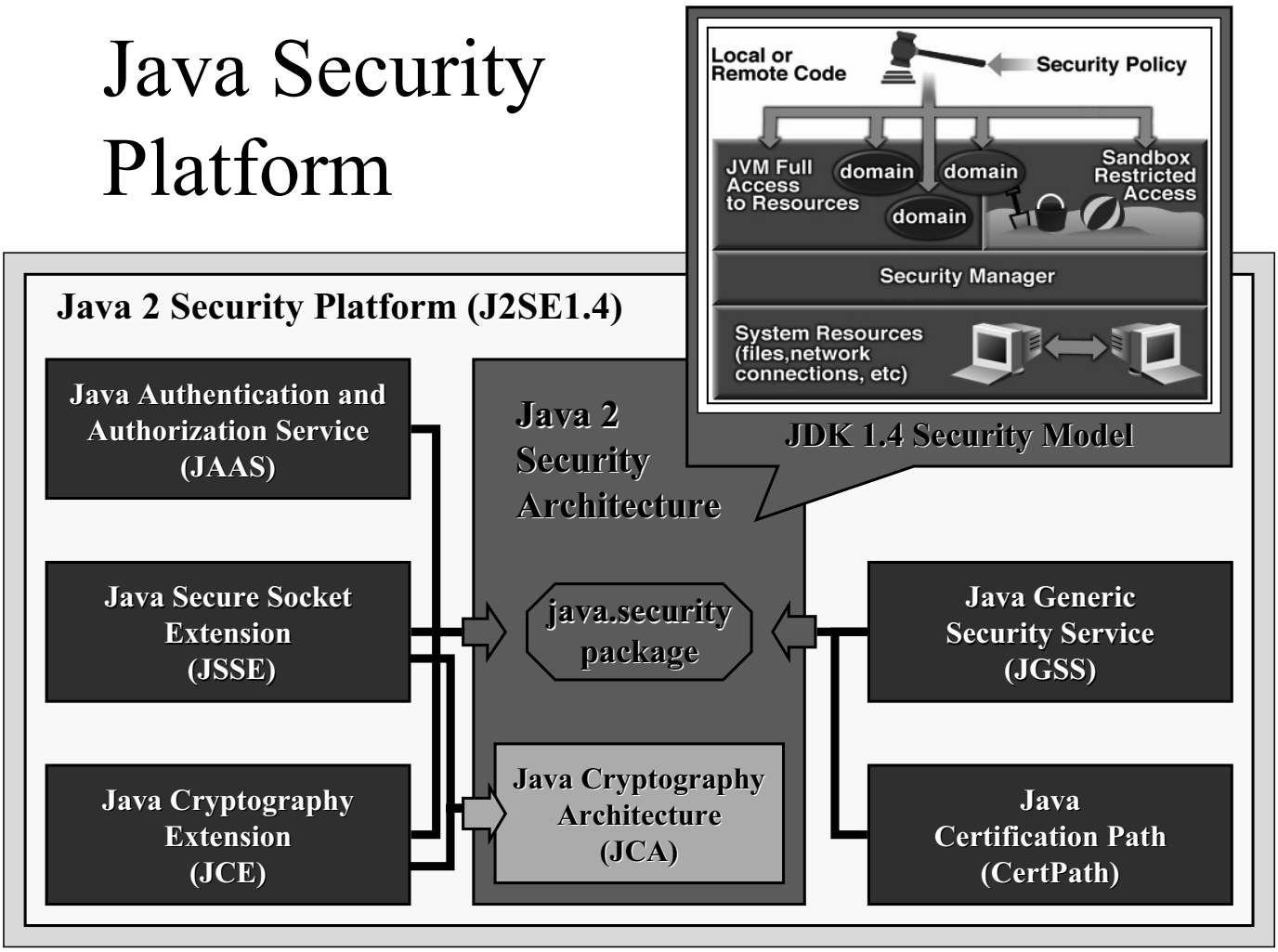
    JCE (Java Cryptography Extension)
    JSSE (Java Secure Socket Extension)
    JAAS (Java Authentication and Authorization Service)

  - 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 dall'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

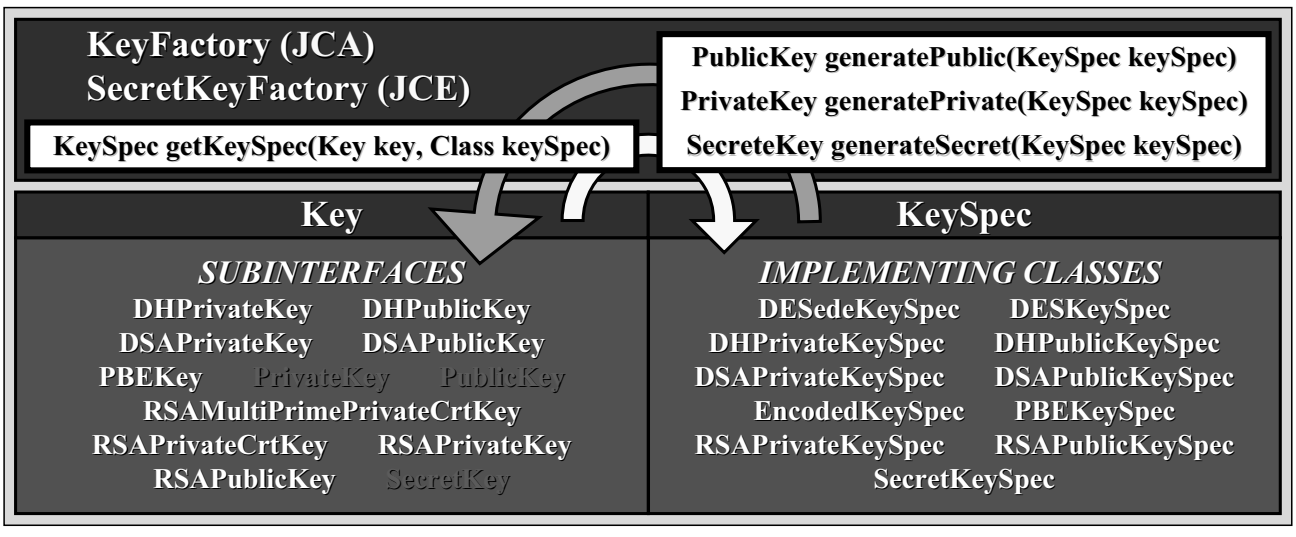
Principali engine classes

- Le seguenti classi di tipo engine sono definite nel package java.security:
  - Key – definisce le funzionalità condivise da chiavi "opache" ("*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 limited access 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 due livelli:
  - 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 di proprietà relativo alla sicurezza "java.security" 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 in 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

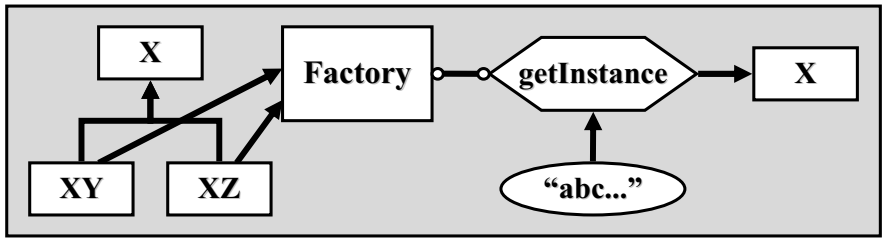
- Le estensioni JCE e JSSE implementano ed estendono le tecniche crittografiche definite dalla JCA fornendo CSP alternativi a "Sun":
  - "**SunJCE**" (crittografia a livello locale)
  - "**SunJSSE**" (crittografia a livello di rete)

- 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")
```

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
- Si basa sui medesimi principi di progetto della JCA
- 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

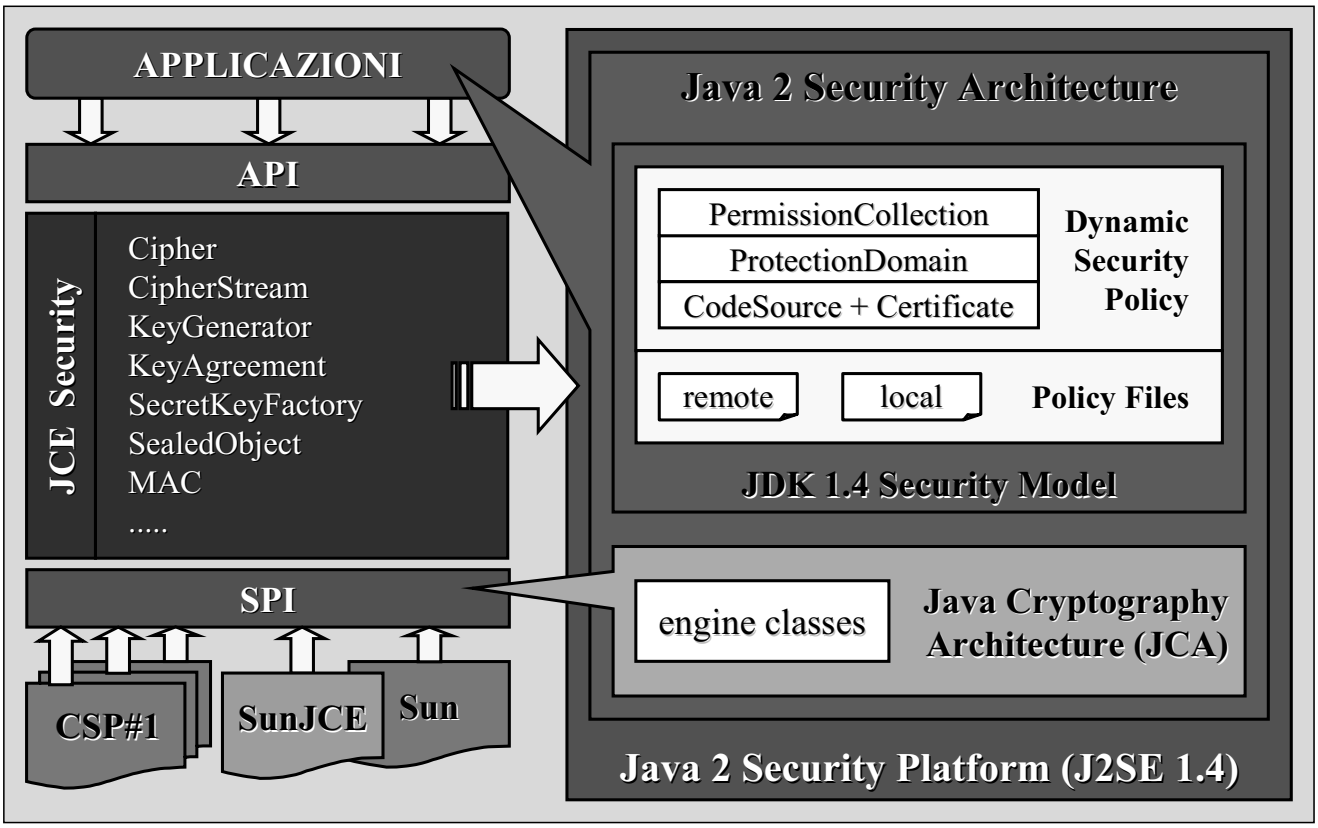
Classi principali

- 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

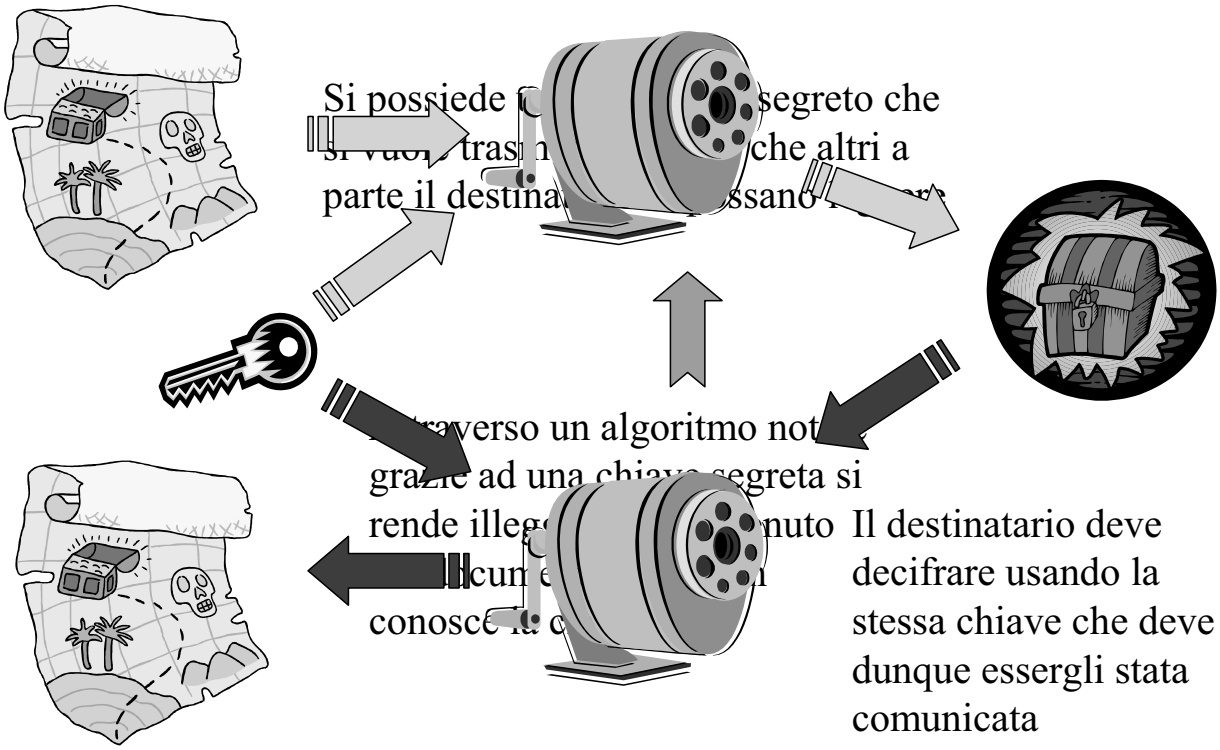
- Il package provider "**SunJCE**" include:
  - una implementazione degli algoritmi di cifratura DES, TripleDES e Blowfish in modalità ECB, CBC, CFB, OFB e PCBC
  - un KeyGenerator per DES, TripleDES, Blowfish, HMAC-MD5, HMAC-SHA1
  - una implementazione di MD5 con DES-CBC PBE definito in PKCS#5
  - 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

Cosa si vuole ottenere:



TripleDES

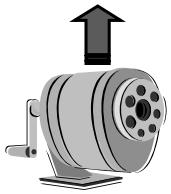
```
import java.security.*;
import javax.crypto.*;

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

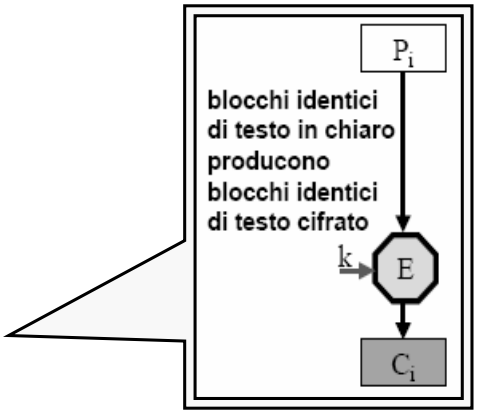
Genera la chiave
Key non viene istanziata con una new

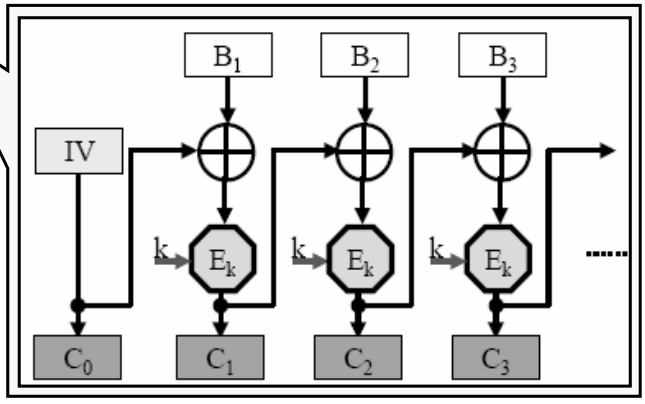


Istanzia il cifrario
Tipo di chiave [TripleDES]
Modalità [ECB]
Padding [PKCS5Padding]

Modalità

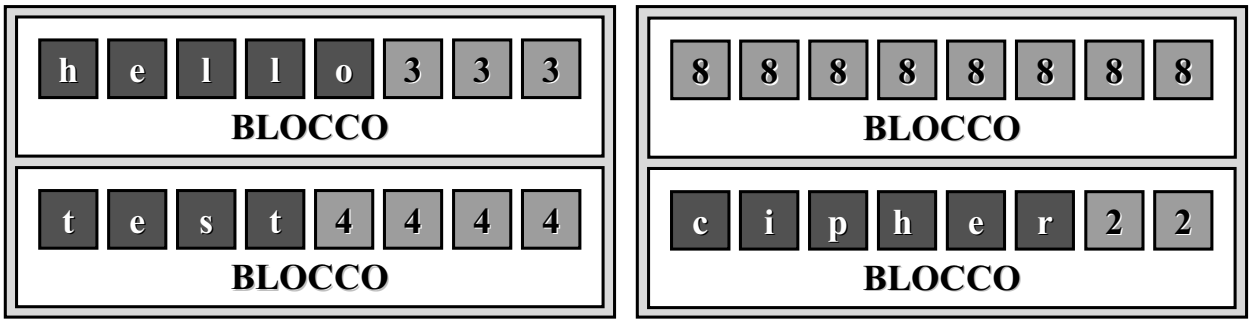
- La modalità definisce come un cifrario debba applicare un algoritmo di cifratura
- Si può specificare se un cifrario deve essere a blocchi o a flusso
- Due tra le modalità più comuni sono:
  - **ECB** [Elettronic Code Book]
    - Lo stesso blocco di testo in chiaro viene sempre cifrato con lo stesso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



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

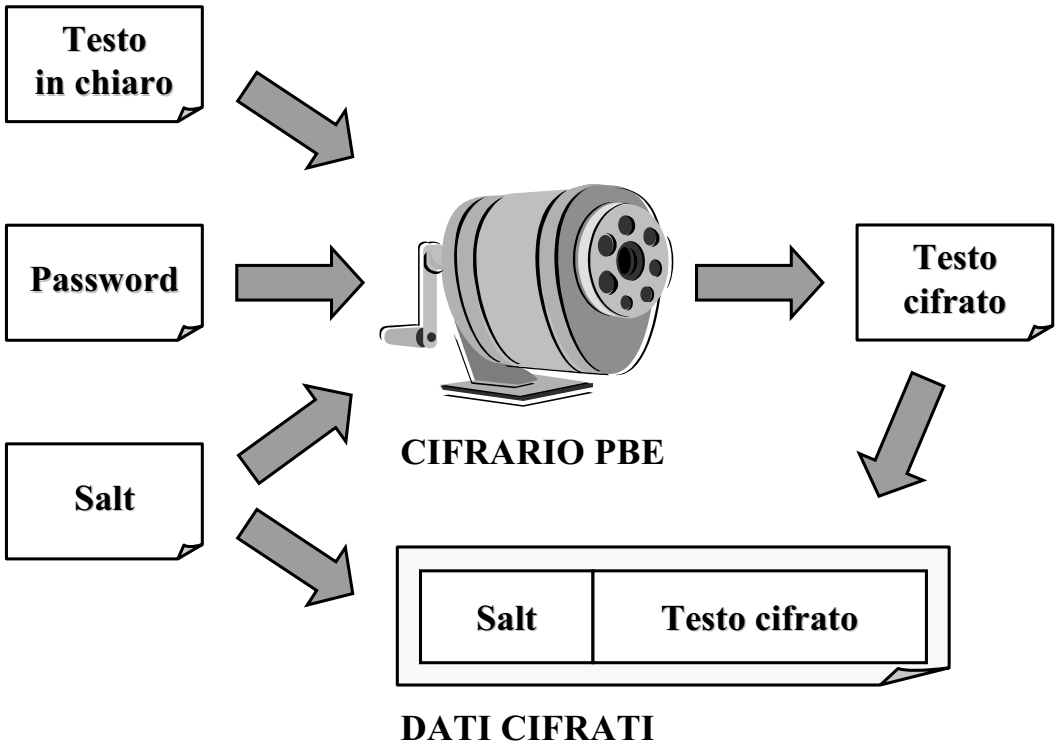
- Il codice visto così com'è non funziona!
  - Occorre infatti gestire le eccezioni...

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

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("UTF8"));
     String saltString = new String(salt);
     String ciphertextString = new String(ciphertext);
     return saltString+ciphertextString;
  } .....
```

Usato per i conteggi di ripetizione

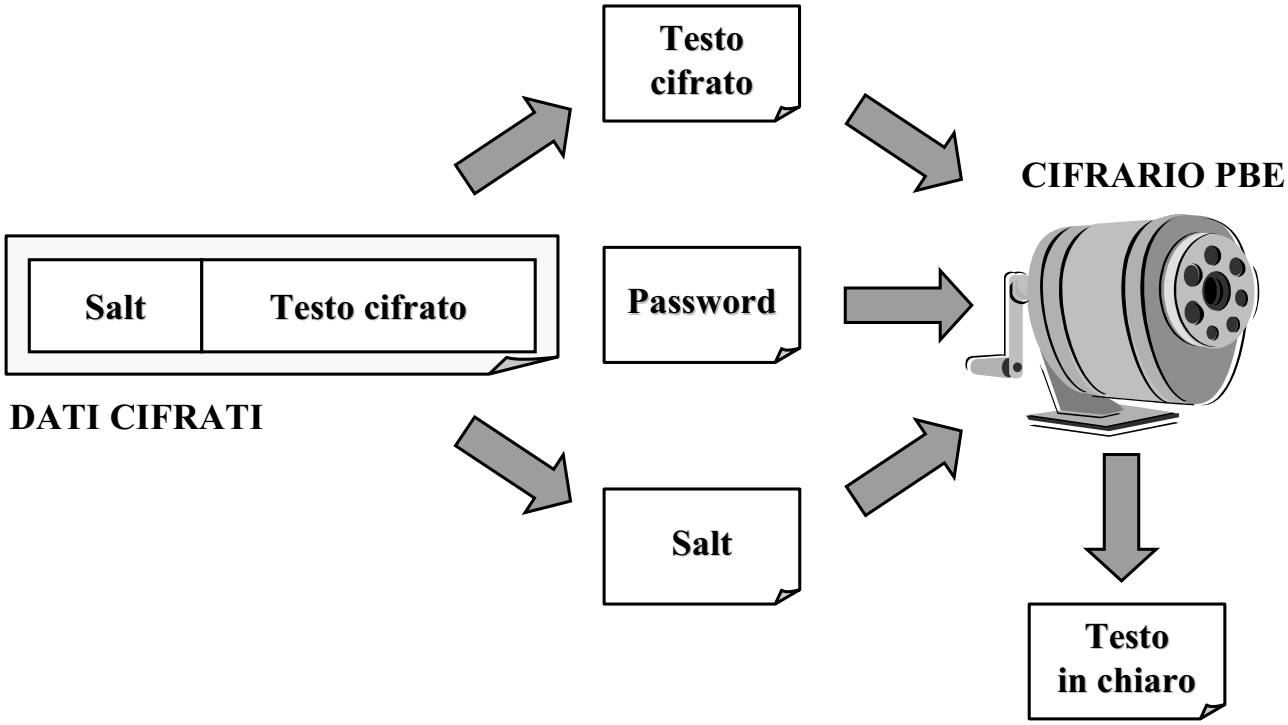
Generazione di un salt casuale

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("UTF8");
  byte[ ] ciphertext = ciphertextString.getBytes("UTF8");

  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 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);
```

- Esempio di utilizzo per la cifratura:

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

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

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 memorizzare e trasportare 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