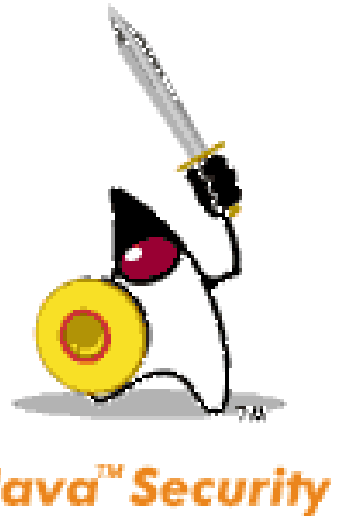
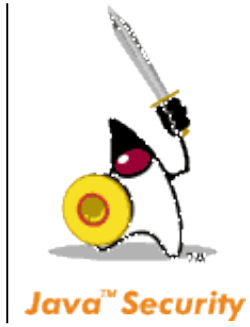
Crittografia & Java Cryptographic Architecture (JCA)

29-30 Settembre
1 Ottobre 2006

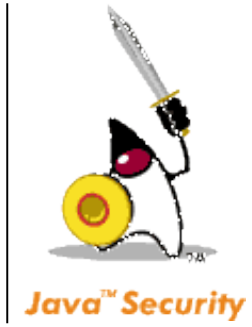


A cura di Franzin Michele

Copyright

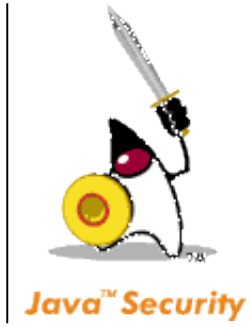


Prerequisiti



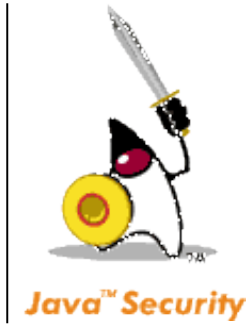
- Saper scrivere un programma in Java.
- Comprendere la cifratura a doppia chiave pubblica-privata.
- Comprendere il concetto di firma digitale.
- Comprendere il concetto di impronta di un messaggio.

Obiettivi



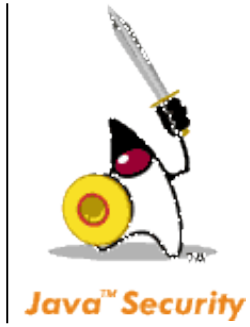
- Acquisire nozioni basilari di crittografia
- Conoscere la Java Cryptographic Architecture (JCA)
- Saper implementare meccanismi di sicurezza in Java

Sommario



- Vengono introdotti i concetti basilari della crittografia, con particolare riferimento a chiavi, algoritmi, cifratura asimmetrica, cifratura asimmetrica, decifratura, digest e firma.
- Viene discussa la Java Cryptographic Architecture (JCA) per l'implementazione della sicurezza e la Java Cryptographic Extension (JCE) con le relative classi.
- Vengono presentate implementazioni in Java: impronta di un messaggio, firma digitale, cifrario a chiave simmetrica.

La crittografia

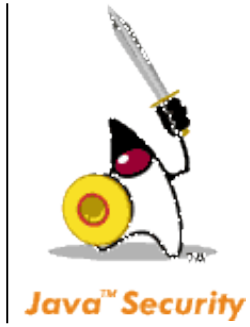


La crittografia è un particolare **processo** grazie al quale, per mezzo di sofisticati algoritmi, è possibile trasformare una sequenza di byte con senso logico (messaggio) in un altra del tutto incomprensibile.

Scopo della crittografia è consentire la trasmissione di un messaggio in forma non intelleggibile ad altri che non sia il destinatario inteso, che deve essere il solo a poterne capire il significato.

La trasformazione avviene grazie ad una ***chiave***: solo chi possiede la chiave per aprire e chiudere il messaggio potrà criptare e decriptare il messaggio.

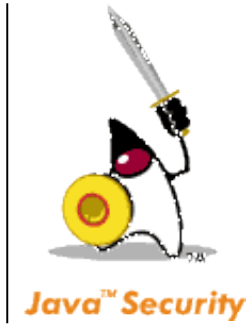
Gli elementi



- Il metodo tramite il quale, dato un ***messaggio in chiaro***, si produce il corrispondente ***messaggio cifrato*** viene chiamato ***algoritmo di cifratura***.

- L'algoritmo deve garantire una proprietà fondamentale: dalla versione cifrata del messaggio deve essere *impossibile risalire al messaggio originale* che l'ha generata.

Algoritmi e chiavi

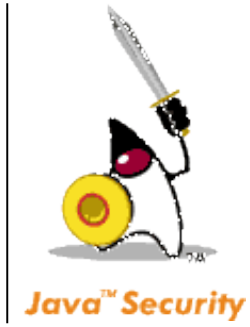


Non è possibile, anzi è spesso inutile e controproducente, basare la sicurezza della comunicazione sulla segretezza dell'algoritmo. Al contrario, *l'algoritmo in sé può anche essere pubblico*: l'importante è che *possa essere in qualche modo "personalizzato"* volta per volta, per uno specifico messaggio da cifrare.

Per questo motivo si distingue fra ***algoritmo di cifratura***, che è quasi sempre pubblico e non varia da un messaggio all'altro, e la ***chiave***, che invece varia spesso, e serve a "personalizzare" caso per caso l'algoritmo.

Grazie a questa "personalizzazione", lo **stesso messaggio**, cifrato con uno ***stesso algoritmo*** di cifratura ma con ***chiavi diverse***, produce ***messaggi cifrati completamente diversi***.

Uno sguardo all'interno ...



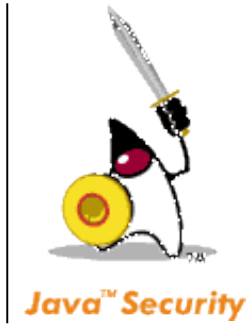
Dunque, indicando con M il messaggio in chiaro, A l'algoritmo di cifratura (pubblico), e K la chiave, il messaggio cifrato M_C può essere espresso come

$$M_C = A(M, K)$$

Per decifrare il messaggio, il destinatario dovrà effettuare l'operazione inversa, che, in generale, richiede di conoscere l'algoritmo A (o la sua controparte di decifratura A', da considerarsi egualmente nota) e la chiave usata per la cifratura:

$$M = A' (M_C, K)$$

... ma funziona ?

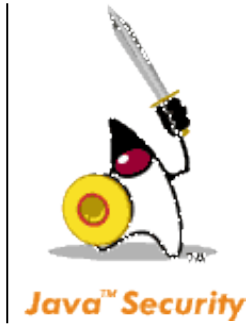


Il solo modo per decifrare il messaggio M_C senza conoscere la chiave K, consiste nel "provare" tutte le chiavi possibili. Per questo motivo, l'utilizzo di chiavi sempre più lunghe (40 bit, 64 bit, 128 bit...) aumenta enormemente il numero di chiavi possibili e rende quindi lunghissime le ricerche esaustive necessarie per "rompere" la cifratura, pur usando molti moderni computer.

Peraltro, dato il continuo aumento di potenza di questi ultimi, chiavi ritenute sicure ieri (es. 40 bit) possono essere ritenute insicure oggi.

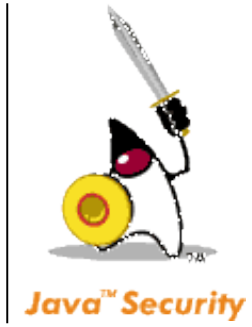
Esistono molti algoritmi, con caratteristiche diverse: in tutti i casi la **robustezza** dipende direttamente dalla lunghezza della chiave K.

Algoritmi a chiave simmetrica ed asimmetrica



- Un algoritmo di cifratura si dice a ***chiave simmetrica*** quando la stessa chiave K viene usata sia per la cifratura, sia per la successiva decifratura.
- Un algoritmo è a ***chiave asimmetrica*** quando cifratura e decifratura richiedono due chiavi diverse.
- Le chiavi sono spesso "imparentate".

Algoritmi a chiave asimmetrica

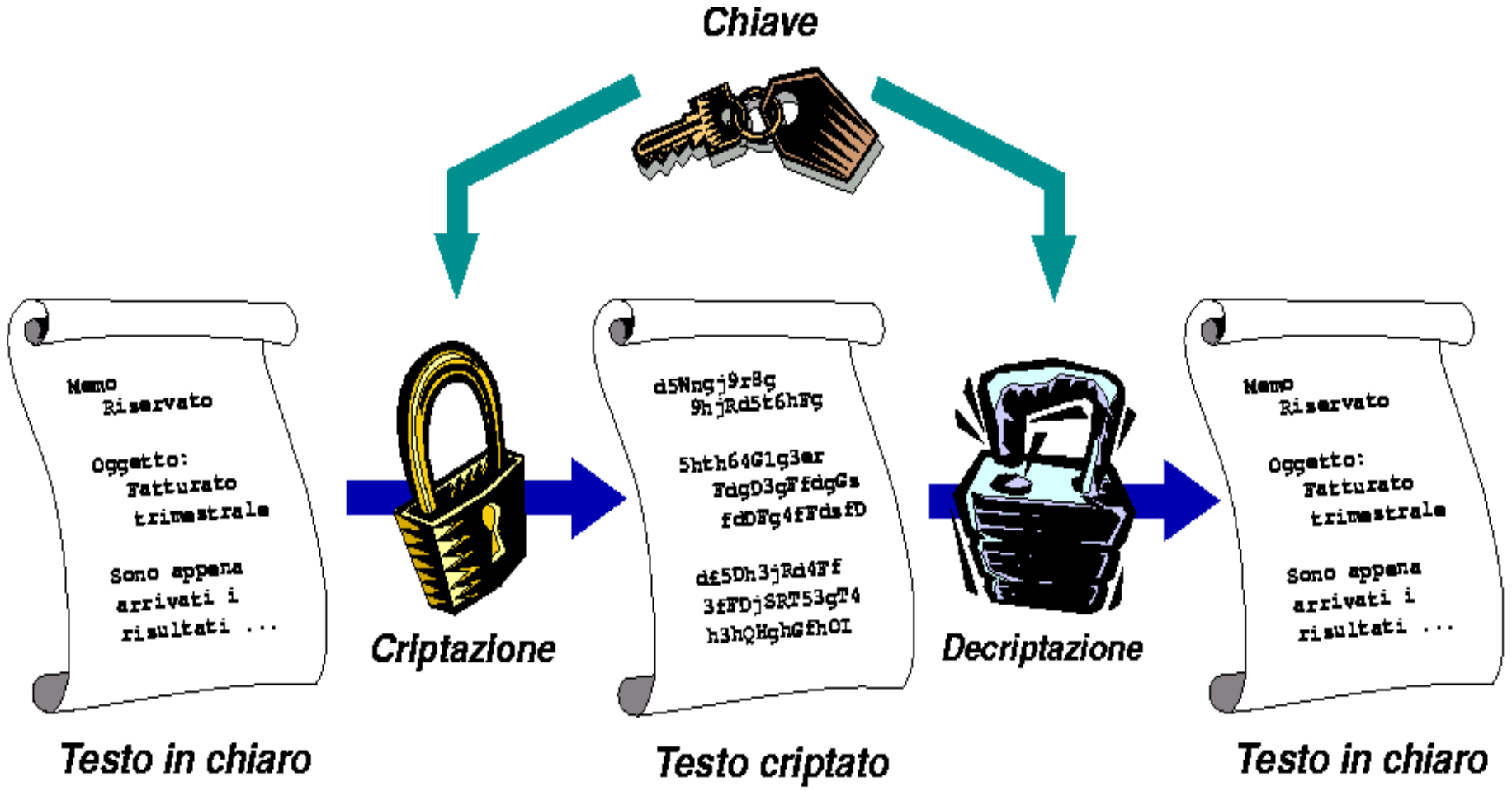


Gli algoritmi a chiave simmetrica sono semplici da realizzare, relativamente efficienti e quindi assai diffusi: fra i più noti vi sono il DES (Data Encryption Standard), triple-DES, IDEA, ecc.

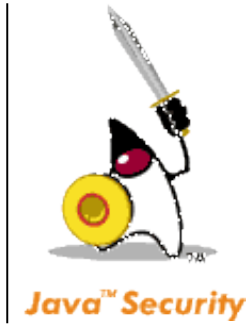
Sfortunatamente, la chiave non può essere trasmessa come un normale messaggio, in quanto essa stessa potrebbe essere intercettata, e ciò renderebbe inutile la cifratura del messaggio.

Perché questo schema funzioni occorre dunque che la chiave possa essere inviata dal mittente al destinatario *in qualche altro modo*.

Crittazione simmetrica



Algoritmi a chiave asimmetrica

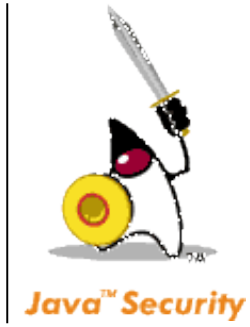


Sebbene con gli algoritmi a chiave asimmetrica la chiave di cifratura non debba essere trasmessa, occorre comunque che il destinatario "conosca" in qualche modo ***l'altra*** chiave, quella di decifratura.

Questo problema è stato risolto da una particolare categoria di algoritmi a chiave asimmetrica, noti come *algoritmi basati su chiave pubblica e chiave privata*.

Non esistono a priori una ***chiave per cifrare*** e una ***chiave per decifrare***: semplicemente, se una delle due chiavi viene usata per cifrare, occorre l'altra per decifrare.

Chiave asimmetriche

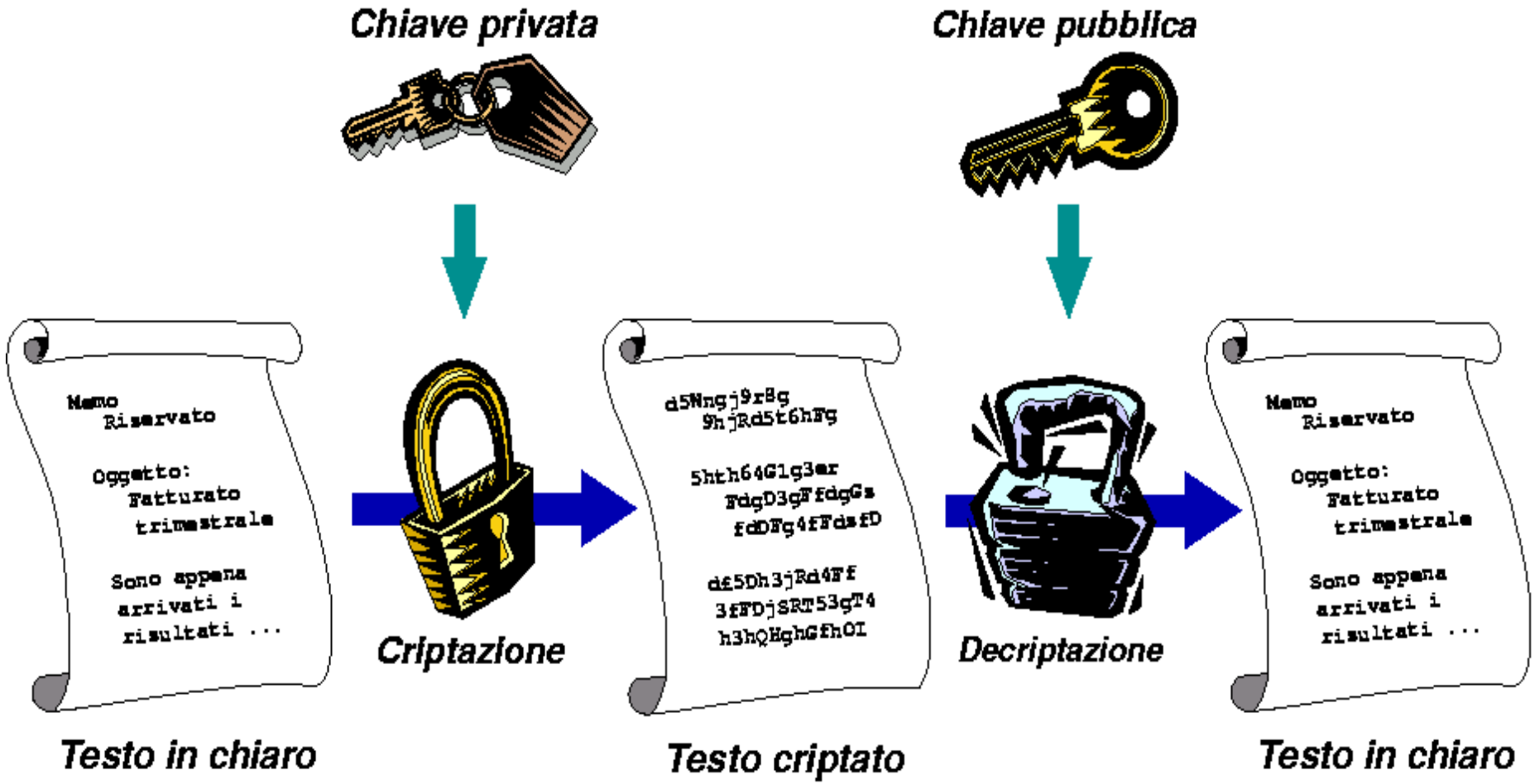


Le due chiavi sono dunque perfettamente intercambiabili: tuttavia, la conoscenza della chiave usata per cifrare ***non è di alcuna utilità*** per decifrare, in quanto per farlo occorre l'altra chiave.

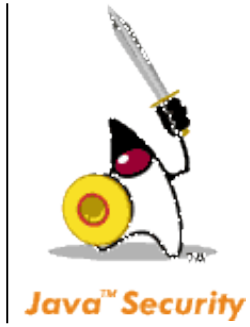
L'algoritmo è strutturato matematicamente in modo che da una chiave sia impossibile risalire all'altra; in questo modo si supera il problema della trasmissione della chiave che affliggeva gli algoritmi a chiave simmetrica.

Chiaramente, la potenza di questo schema sta nel non richiedere la trasmissione "sicura" della password - anzi, una delle due è talmente poco sicura da poter essere pubblicata.

Crittazione asimmetrica



Identificare il destinatario



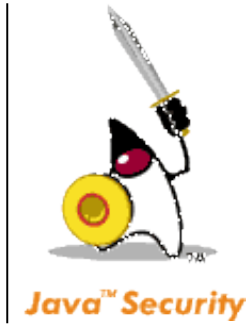
Se A vuole inviare un messaggio a B *con la certezza che solo B lo possa leggere*, basta che lo cripti usando *la chiave pubblica di B* (che è disponibile proprio in quanto pubblica: si può pensare di recuperarla da appositi elenchi, non diversi dagli elenchi telefonici).

Dopo di ciò, solo B potrà leggere il messaggio cifrato, in quanto è il solo a disporre della chiave privata necessaria per la decifratura.

Chiunque intercetti il messaggio non potrà decifrare nulla, neppure procurandosi la chiave pubblica di B (che pure è disponibile), in quanto quest'ultima è incapace di decifrare un messaggio da lei stessa cifrato.

NB: B non può avere alcuna certezza che l'autore del messaggio sia proprio A: infatti, chiunque può prendere la chiave pubblica di B e inviargli un messaggio.

Indentificare il mittente

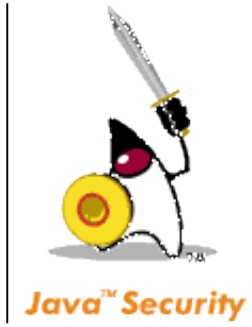


Se A vuole inviare un messaggio a B *con la certezza che sia stato mandato da A*, basta che lo cifri usando *la sua chiave privata (di A).*

In questo modo, chiunque potrà leggere il messaggio cifrato (perché la chiave pubblica di A necessaria per la decifratura è liberamente disponibile), ma nel farlo avrà anche la certezza che *solo A può esserne l'autore*, in quanto solo A poteva conoscere la chiave privata di A che ***certamente*** è stata usata per la cifratura.

Questa certezza deriva dal fatto che il messaggio si decifra con la chiave pubblica di A.

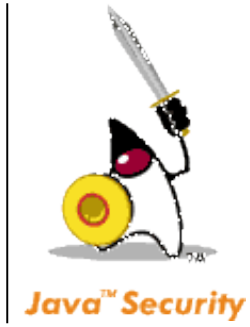
Comunicazioni private



Se A vuole inviare un messaggio a B *con la certezza che solo B lo possa leggere **e** inoltre comprovando che l'autore è proprio A*. Il messaggio M verrà cifrato una prima volta usando *la **chiave privata di A***, e subito dopo una seconda volta usando ***la chiave pubblica di B*** (l'ordine di questi due passaggi è irrilevante).

Ovviamente, tale messaggio è incomprensibile per chiunque lo intercetti. Ma soprattutto, per ricostruire il messaggio in chiaro B dovrà decifrare due volte il messaggio ricevuto: una prima volta usando la sua chiave privata (e ciò fa sì che solo B possa leggere il messaggio), e poi una seconda usando la chiave pubblica di A (il che comprova che proprio A ne è l'autore).

Digest di messaggi



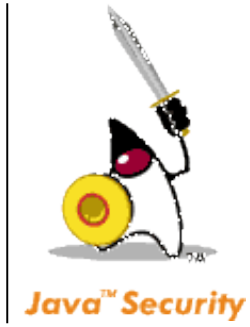
Un DIGEST (o hash) di un messaggio è una particolare versione cifrata del messaggio, caratterizzata dal fatto di avere *dimensione fissa* indipendente dalla dimensione del messaggio originale.

Gli algoritmi usati per calcolare il digest devono assicurare che sia *"estremamente improbabile" che due messaggi diversi possano condurre alla medesima impronta*.

Grazie a questa proprietà, i digest possono essere usati per *identificare in modo certo e univoco* i dati del messaggio, di cui costituiscono a tutti gli effetti una sorta di "impronta digitale".

Questa tecnica è spesso usata per *"garantire l'autenticità" di oggetti*, nel senso di assicurare chi li riceve che essi non sono stati alterati.

Signature di messaggi

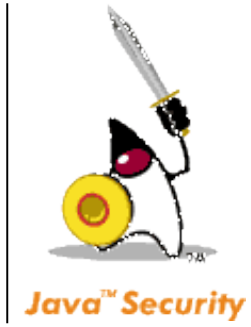


Una SIGNATURE (o firma) di un messaggio è una ***stringa cifrata***, spesso relativamente corta e di lunghezza fissa, ottenuta dal messaggio originale tramite un algoritmo e una chiave (privata).

Questa stringa viene usata per *firmare i dati*: a tale scopo è solitamente trasmessa insieme ai dati da cui è stata ricavata e di cui costituisce la firma digitale.

Tale firma può poi essere *verificata* (tipicamente dal destinatario) applicando ai dati e alla firma ricevuta l'algoritmo di verifica, unitamente alla chiave pubblica del mittente.

L'architettura crittografica di Java (JCA)



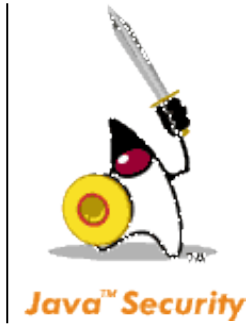
E' ispirata a due principi:

- Indipendenza dall'implementazione e interoperabilità
- Estendibilità e indipendenza degli algoritmi

Tradotti nei seguenti criteri:

- Una ***engine class*** è una classe che definisce le funzionalità di un dato tipo di algoritmo crittografico, senza però fornire alcuna implementazione (classi astratte). Ad esempio, MessageDigest, Signature e KeyPairGenerator sono tre diverse engine class, e definiscono rispettivamente le funzionalità attese dagli algoritmi di tipo Message Digest, Signature, e Generatori di coppie di chiavi.
- Un ***provider*** ("fornitore") è un package che fornisce l'implementazione concreta di un certo insieme di funzionalità crittografiche. Più provider, di diversi produttori, possono coesistere e interoperare.

JCA & JCE



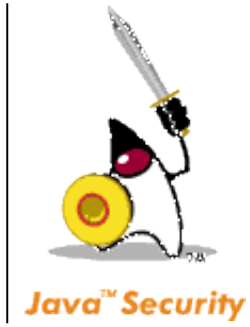
L'architettura crittografica di Java in versione base (JCA) definisce quindi il framework generale per la crittografia, lasciando alla **Java Cryptographic Extension (JCE)** il compito di fornire l'implementazione completa delle funzionalità di cifratura e decifratura.

La JCE della Sun non è esportabile fuori dagli USA !

Date le restrizioni all'esportazione di codice crittografico in forma binaria dagli USA, il codice standard Sun per la Java Security e per la Java Cryptographic Extension non può essere esportato (né scaricato via Internet !) dagli Stati Uniti.

Occorre procurarsi librerie equivalenti prodotte in Europa (o comunque fuori dagli USA), come Cryptix (Internazionale) e IAIK (Austria).

Security Provider



Il provider di default si chiama SUN, ed è built-in nel JDK.

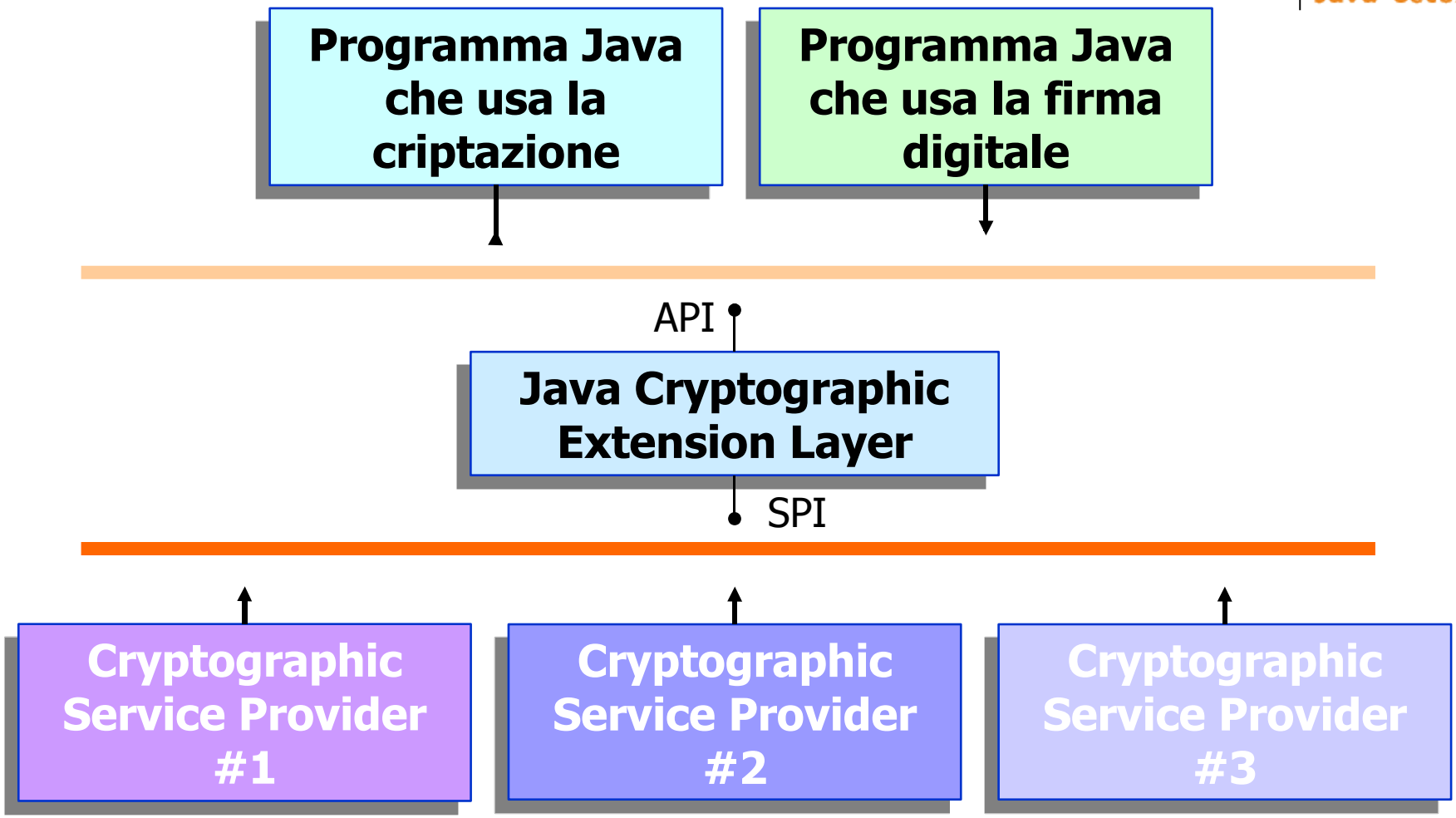
Perciò, un provider potrebbe fornire, ad esempio, una implementazione dell'algoritmo DSA (Digital Signature Algorithm) e/o dell' RSA Cryptosystem.

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

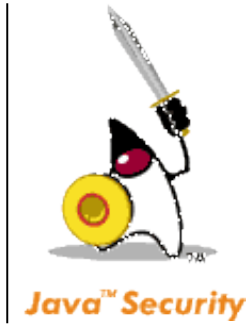
Ogni JDK può avere uno o più provider installati: altri possono essere aggiunti dinamicamente, tramite il metodo

```
Security.addProvider()
```

L'architettura



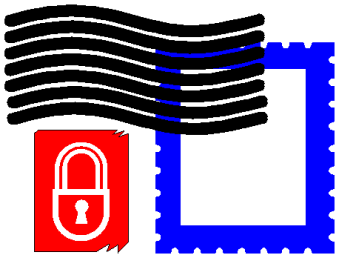
Funzionalità fornite dalla JCA



- Nella general release della JCA, sono fornite
  - le classi `Provider` e `Security`
  - le engine class `MessageDigest`, `Signature`, `KeyPairGenerator`
  - le classe `Key` e le classi collegate
- Pertanto, con tali funzionalità di base si possono solo calcolare impronte di messaggi e firme digitali di messaggi, usando coppie di chiavi pubblica/privata. Non è invece possibile, senza un provider JCE, cifrare e decifrare messaggi in modo completo (con algoritmi più o meno noti, quali ad esempio DES, IDEA, ecc).
- Algoritmi supportati:
  - MessageDigest supporta gli algoritmi SHA-1, MD2, MD5
  - Signature supporta gli algoritmi DSA e RSA-based (quest'ultimo con SHA-1, MD2 o MD5)
  - KeyPairGenerator supporta gli algoritmi di generazione per DSA e RSA

Casi d'uso

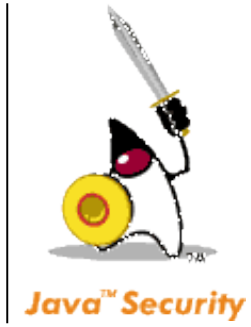
2) Calcolo dell'impronta digitale di un messaggio.



4) Generazione di una coppia di chiavi pubblica/privata e suo uso in una Signature.

6) Creazione di un Cipher per cifrare un messaggio (algoritmo a chiave simmetrica).

Calcolo dell'impronta digitale di un messaggio

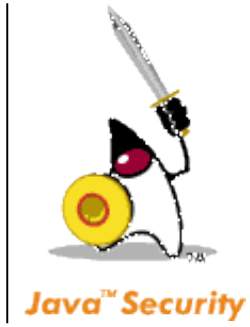


- Si crea un oggetto `MessageDigest` specializzato per un dato algoritmo (fra quelli supportati).

- Si forniscono i dati da processare tramite il metodo `update()`: un messaggio fatto di più dati può essere trattato tutto insieme, dopo aver inserito tutti i dati nell'oggetto `MessageDigest` tramite successive invocazioni di `update()`.

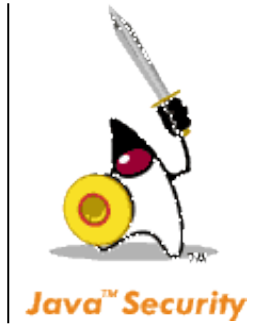
- Si produce l'impronta col metodo `digest()`.

Inizializzazione



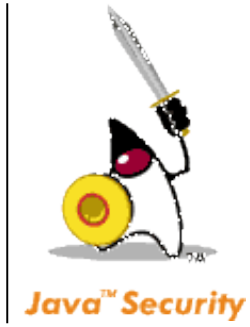

```
import java.security.*;

class Hashing {

public static void main(String args[]){
   MessageDigest sha = null;
   try {
      // fa creare un nuovo oggetto MessageDigest
      // specializzato per l'algoritmo richiesto
      // (SHA = Secure Hash Algorithm)
      sha = MessageDigest.getInstance("SHA-1");
   } catch (NoSuchAlgorithmException e) {
      System.out.println("Algoritmo richiesto non " +
                         "supportato");
   }
```

Generazione dell'impronta



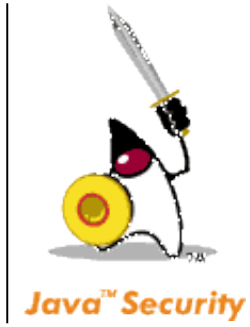

```
    byte[] messaggio1a = {10,20,30,40,50};
    byte[] messaggio1b = {60,70,80,90};
    sha.update(messaggio1a);
    sha.update(messaggio1b);
    // hash è messaggio1a + messaggio1b cifrato
    byte[] hash1 = sha.digest();

    // più rapido, per un messaggio singolo
    byte[] messaggio2 = {71,72,73,74,75,76,77,78,79,80};
    byte[] hash2 = sha.digest(messaggio2);

    System.out.println("Digest di messaggio1 (a+b): " +
                       hash1);
    System.out.println("Digest di messaggio2: " +
                       hash2);
    }
}
```

Generazione di una coppia di chiavi e uso in una Signature



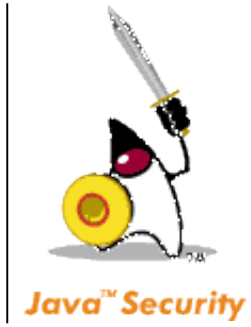
- Generazione della coppia di chiavi
  - si crea un oggetto `KeyPairGenerator` specializzato per un dato algoritmo (fra quelli supportati)
  - lo si inizializza con un seme casuale (`SecureRandom`)
  - gli si fa generare una coppia di chiavi (pubblica e privata)
  - si recuperano singolarmente le due chiavi (due oggetti di tipo `PublicKey` e `PrivateKey`)
- Generazione della firma
  - si crea un oggetto `Signature` specializzato per un dato algoritmo (fra quelli supportati)
  - lo si inizializza con la chiave *privata* (`initSign()`)
  - gli si forniscono i dati da firmare (`update()`)
  - gli si fa generare la firma (`sign()`)
- Verifica della firma
  - si re-inizializza l'oggetto `Signature` con la chiave *pubblica* (`initSign()`)
  - gli si forniscono i dati da verificare (`update()`)
  - gli si fa verificare la firma (`verify()`)

Generazione chiavi pubblica & privata



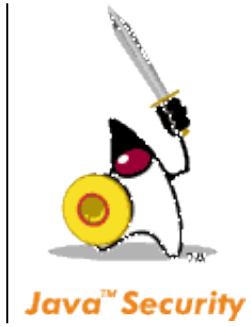

```
public static void main(String args[]){
   KeyPairGenerator gen = null;

   try {
      // DSA = Digital Signature Algorithm
      gen = KeyPairGenerator.getInstance("DSA");
   } catch (NoSuchAlgorithmException e1) {
      System.out.println("Algoritmo non supportato");
      System.exit(1);
   }

   // Chiavi modulo 1024 bit, seme definito da Java
   gen.initialize(1024, new SecureRandom( ));
   // coppia di chiavi pubblica/privata
   KeyPair kp = gen.generateKeyPair();
   // recupera le due chiavi, pubblica e privata
   PrivateKey kpriv = kp.getPrivate();
   PublicKey kpub = kp.getPublic();
```

Generazione signature



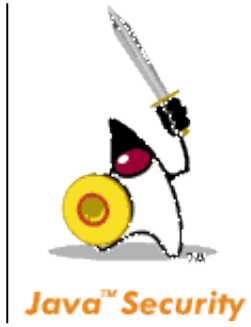

```
Signature s = null;

try {
   // genera un nuovo oggetto Signature...
   s = Signature.getInstance("DSA");
} catch (NoSuchAlgorithmException e2) {
   System.out.println("Algoritmo non supportato");
   System.exit(2);
}

try {
   // ... e lo inizializza con la chiave privata
   s.initSign(kpriv);
} catch (InvalidKeyException e3) {
   System.out.println("Chiave privata non valida");
   System.exit(3);
}
```

Generazione della firma



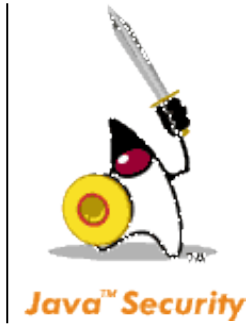

```
byte[] data = {10,20,30,40,50,60,70,80,90};
byte[] firma = null;

try {
   s.update(data);
   // firma contiene la versione firmata di data
   firma = s.sign();
} catch (SignatureException e4) {
   System.out.println("Firma non valida");
   System.exit(4);
}

System.out.println("Dati: " + data);
System.out.println("Firma: " + firma);
```

Verifica della firma



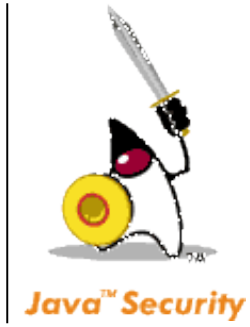

```
try {
   // inizializza la Signature con la chiave di decifratura
   s.initVerify(kpub);
} catch (InvalidKeyException e5) {
   System.out.println("Chiave pubblica non valida");
   System.exit(5);
}

boolean res = false;
try {
   // verifica la firma
   s.update(data);
   res = s.verify(firma);
} catch (SignatureException e6) {
   System.out.println("Firma non valida in Signature");
   System.exit(6);
}

System.out.println("Esito della verifica: " + res);
```

Creazione di un Cipher per cifrare un messaggio 1



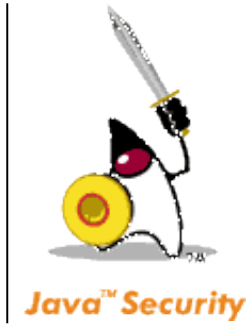
- Creazione e installazione del provider IAIK
  - si crea un nuovo oggetto provider IAIK e lo si aggiunge all'elenco gestito dalla classe `Security`
- Generazione della chiave appropriata per l'algoritmo richiesto
  - si crea un oggetto `KeyGenerator` specializzato per un dato algoritmo (fra quelli supportati), precisando il provider desiderato (IAIK)
  - lo si inizializza con un seme casuale (un oggetto `SecureRandom`)
  - gli si fa generare la chiave richiesta (restituisce un oggetto di tipo `SecretKey`)
  - si recupera la chiave (un oggetto di tipo `SecretKey`)

Creazione di un Cipher per cifrare un messaggio 2



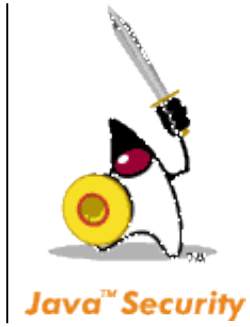
- Creazione del Cipher e generazione del messaggio cifrato
  - si crea un oggetto `Cipher` specializzato per un dato algoritmo (fra quelli supportati), precisando altresì il provider richiesto (IAIK)
  - lo si inizializza in modalità ENCRYPT con la chiave segreta (in IAIK-JCE, tramite `init()`; altre specifiche prevedono invece che tale metodo si chiami `initEncrypt()`)
  - gli si fa generare la versione cifrata del messaggio (in IAIK-JCE, tramite `doFinal()`; altre specifiche prevedono invece che tale metodo si chiami `crypt()`)
- Decifratura del messaggio e verifica del risultato
  - si re-inizializza l'oggetto Cipher con la medesima chiave segreta, ma in modalita' DECRYPT (in IAIK-JCE, tramite `init()`; altre specifiche prevedono invece che tale metodo si chiami `initDecrypt()`)
  - gli si fa decifrare il messaggio (in IAIK-JCE, tramite `doFinal()`; altre specifiche prevedono invece che tale metodo si chiami `decrypt()`)

Creazione del provider



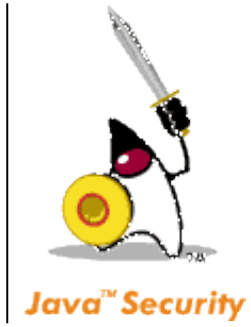

```
import java.security.*;
import javax.crypto.*;
import iaik.security.provider.*;

class Simmetrica {

static void printBytes(String name, byte[] b){
   System.out.print(name + ": ");
   for (int i=0; i<b.length; i++)
      System.out.print(b[i] + " ");
   System.out.println("");
   System.out.println("-----------------------");
}

public static void main(String args[]){
   IAIK provider = new IAIK();
   Security.addProvider(provider);
```

Generazione chiave



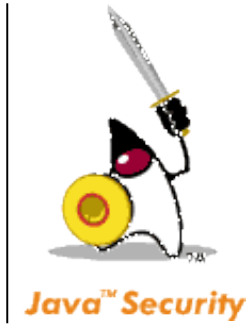

```
KeyGenerator gen = null;
try {
   gen = KeyGenerator.getInstance("DES","IAIK");
} catch (NoSuchAlgorithmException e1) {
   System.out.println("Algoritmo non supportato");
   System.exit(1);
} catch (NoSuchProviderException e1) {
   System.out.println("Provider non supportato”);
   System.exit(1);
}

gen.init(new SecureRandom());
// Sun: gen.initialize()

Key k = gen.generateKey();
Cipher des = null;
byte[] data = {10,20,30,40,50,60,70,80};
byte[] dataK = null, newData = null;
```

Cifratura e decifratura



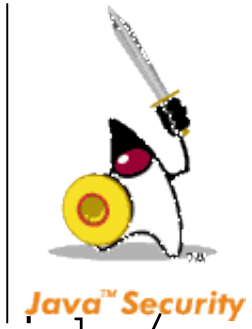

```
// Creazione Cifrario
des = Cipher.getInstance("DES","IAIK");

// Inizializzazione per cifratura
des.init(Cipher.ENCRYPT_MODE, k);
// Sun: des.initEncrypt(k);

// Cifratura
dataK = des.doFinal(data);
// Sun: dataK = des.crypt(data);

// Inizializzazione per decifratura
des.init(Cipher.DECRYPT_MODE, k);
// Sun: des.initDecrypt(k);

// Decifratura
newData = des.doFinal(dataK);
// Sun: newData = des.crypt(dataK);
```

Verifica




```
boolean res = true;

if (data.length != newData.length)
   res = false;
else
   for (int j=0; j<data.length; j++)
      if (data[j] != newData[j]) {
         res = false;
         break;
      }

System.out.println("Esito della verifica: " + res);
```

Risorse



Java Cryptography Architecture (JCA)

`http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html`

Java Cryptographic Extension (JCE)

`http://java.sun.com/j2se/1.5.0/docs/guide/security/jce/JCERefGuide.html`

The IAIK Java Cryptographic Extension

`http://jce.iaik.tugraz.at/`

Cryptix toolkit

`http://www.cryptix.org/`

Bouncy Castle

`http://www.bouncycastle.org/`

Informazioni sul JUG Padova

Sito Web

`http://www.jugpadova.it`

Mailing List

`http://groups.yahoo.com/group/JUG_Pado`

Persone di riferimento

- Dario Santamaria (dario.santamaria@jugpadova.it)
- Lucio Benfante (lucio.benfante@jugpadova.it)
- Paolo Donà (paolo.dona@jugpadova.it)