Crittografia in Java

JCA e JCE

- **Java Cryptography Architecture (JCA)**
  - JCA Reference Guide:
    http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html
  - Fa parte di JDK 2
  - Supporta funzioni hash, firma digitale, random number generator
- **Java Cryptography Extension (JCE)**
  - Estende JCA con i cifrari

JCA principles

- implementation independence
  - security services are accessible through a standard interface implemented by *providers*
- implementation interoperability
  - an application is not bound to a specific provider, and a provider is not bound to a specific application.
- algorithm extensibility
  - The Java platform supports the installation of custom providers that implement such services

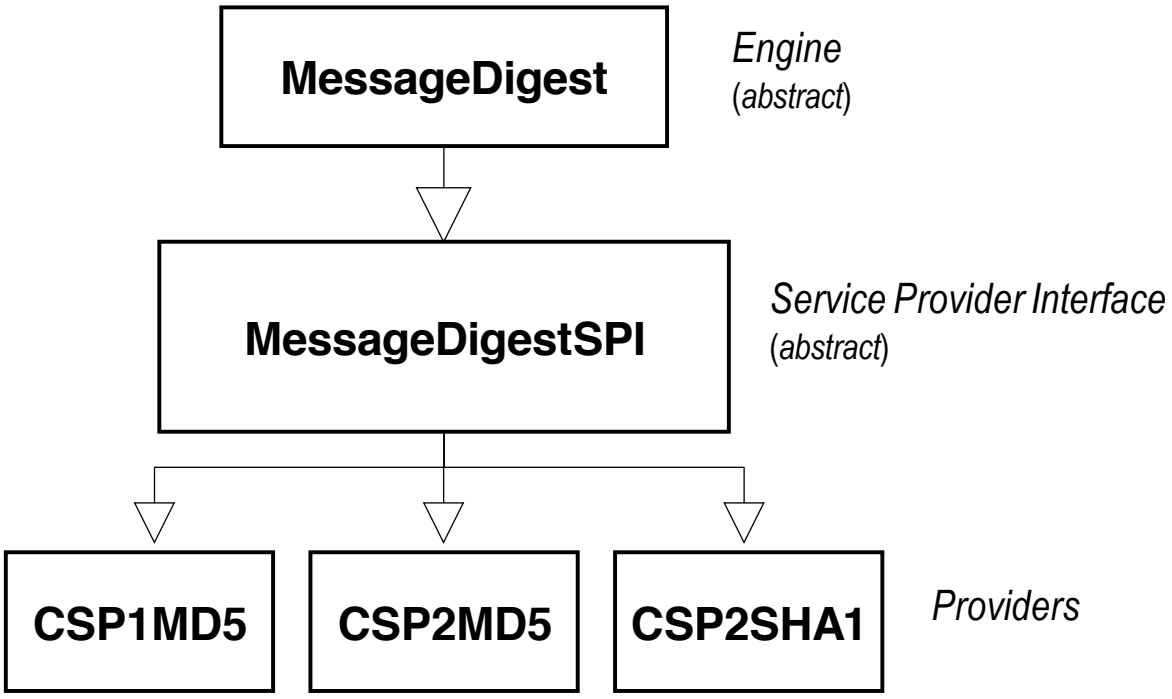
Principali classi di JCA

- **MessageDigest**
- **Signature**
- **KeyPairGeneration**
- **KeyFactory**
- **CertificateFactory**
- **KeyStore**
- **AlgorithmParameters**
- **AlgorithmParameterGenerator**
- **SecureRandom**

Architettura JCA

- **Factory pattern**
  - Definisce un'interfaccia per la creazione di un oggetto ma lascia che le sottoclassi decidano quale classe istanziare
  - Un'istanza si ottiene con **getInstance()**
- **Strategy pattern**
  - Astrae una famiglia di algoritmi, li incapsula e li rende interscambiabili, permettendo all'utente di scambiare gli algoritmi provenienti da provider diversi

Architettura JCA



Provider

- **Un provider è un insieme di implementazioni di vari algoritmi**
- Java 2 incorpora almeno il provider SUN (sun.java.security.sun)
  - MD5, SHA-1
  - DSA: firma, verifica e generazione delle chiavi, parametri
  - Creazioni di certificati X.509
  - Generazione di numeri random proprietaria
  - Keystore proprietario
- Altri provider RSAJCA (com.sun.rsajca.Provider)
  - Gestione chiavi RSA
  - Firma digitale RSA con SHA-1 o MD5

JCE

- JCE ha la stessa architettura di JCA
- JCE è costituita da **javax.crypto** e sotto-package
- Classi principali
  - **Cipher**
  - **KeyAgreement**
  - **Mac**
  - **SecretKey**
  - **SecretKeyFactor**

Principali Provider per JCE

NOME	URL	LICENZA	NOTE
Bouncy Castle	**http://www.bouncycastle.org**	Open Source Apache-style	Provider più completo, free
SunJCE	**http://www.cryptix.org**	Open Source Apache-style	Implementazione di riferimento; non compatibile con altri provider; non supporta RSA

Installazione di un provider

- **Passo 1. Download del provider**
- **Passo 2. Copia del JAR**
  - ***Si consiglia di installare il provider come estensione***
  - Creare il JAR (se necessario)
  - Copiare il JAR in **lib/ext** di JRE
- **Passo 3. Configurazione di java.security e java.policy per abilitare il provider**
- **Passo 4. Collaudo della installazione**

Seguire attentamente le istruzioni fornite dal provider

Cifratura simmetrica

- Encryption in ECB and CBC mode
- CipherStreams
- SealedObjects
- Password-Based Encryption
- Key storage

Cifratura: classi ed interfacce

- **Le principali classi ed interfacce**
  - **javax.crypto.Cipher**
    - **getInstance, init, update, doFinal**
  - **java.security.Key (interfaccia)**
    - Un oggetto **Key** viene creato con un factory **javax.crypto.KeyGenerator** oppure **java.security.KeyFactory**
  - **javax.crypto.KeyGenerator**
    - **getInstance, init, generateKey**

Cifratura: passi

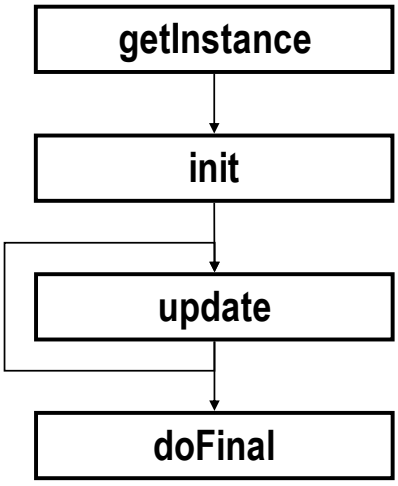
- Creazione di una chiave di cifratura
  1. KeyGenerator keyGenerator = KeyGenerator.getInstance("Blowfish");
  2. keyGenerator.init(128);
  3. SecretKey key = keyGenerator.generateKey();
- Creazione ed inizializzazione di un cifrario
  1. Cipher cipher = Cipher.getInstance("Blowfish/ECB/PKCS5Padding");
- Cifratura
  1. cipher.init(Cipher.ENCRYPT_MODE, key);
  2. byte[] cipherText = cipher.doFinal(stringToEncrypt.getBytes());

(continua)

Cifratura: passi

- Decifratura
  1. cipher.init(Cipher.DECRYPT_MODE, key);
  2. byte[] plainText = cipher.doFinal(cipherText);

La classe Cipher



"algorithm/mode/padding", "algorithm"

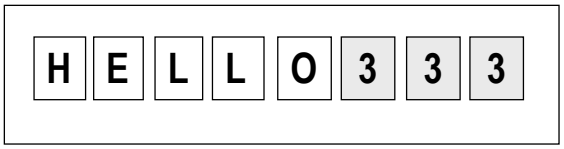
ENCRYPT_MODE, DECRYPT_MODE

Multiple part encryption

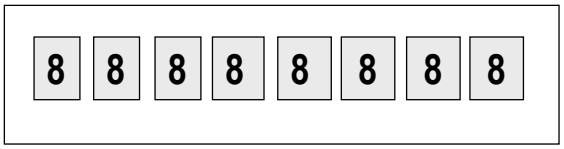
Finishes a multiple part encryption

Padding: lo standard PKCS#5

Il padding è necessario quando il testo in chiaro non è un multiplo del blocco



Ai byte di padding si assegna il numero di byte necessari per colmare un blocco



Se il testo in chiaro è un multiplo di un blocco, si aggiunge un blocco di padding

Il padding viene <u>sempre</u> inserito

Cifratura in modalità CBC

- **Creazione di una chiave di cifratura**
  1. KeyGenerator keyGenerator = KeyGenerator.getInstance("DES");
  2. SecretKey key = keyGenerator.generateKey();
- **Creazione ed inizializzazione di un cifario**
  1. byte[] randomBytes = new byte[8]; // iv size = block size
  2. SecureRandom random = new SecureRandom();
  3. random.nextBytes(randomBytes);
  4. IvParameterSpec ivparams = new IvParameterSpec(randomBytes);
  5. Cipher cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
  6. cipher.init(Cipher.ENCRYPT_MODE, key, ivparams);

Cifratura in modalità CBC

- **Decifratura**
  1. cipher.init(Cipher.DECRYPT_MODE, key, ivparams);
  2. byte[] plainText = cipher.doFinal(cipherText);

Una soluzione alternativa

- **Ricavare i parametri dal cifrario**
  1. AlgorithmParameters params = cipher.getParameters();
- **Decifratura**
  1. cipher.init(Cipher.DECRYPT_MODE, key, params);
  2. byte[] plainText = cipher.doFinal(cipherText);

CipherStreams

- Gli stream **CipherInputStream** e **CipherOutputStream** permettono di cifrare in modo trasparente gli stream avvolti.
- Possono essere utilizzati ovunque si usano **InputStream** e **OutputStream** (file, connessioni di rete).
- Nella **new** si specifica il cifrario da utilizzare

CipherStreams: example

- **Cifratura**

```
1. Cipher cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
2. cipher.init(Cipher.ENCRYPT_MODE, key, ivparams);
3. FileInputStream input = new FileInputStream(plaintextName);
4. FileOutputStream output = new FileOutputStream(ciphertextName);
5. CipherOutputStream cipherOutput = new CipherOutputStream(output, cipher);
6. int r = 0;
7. while ((r = input.read()) != -1)
8.    cipherOutput.write(r);
9. cipherOutput.close();
10. output.close();
11. input.close();
```

CipherStreams: example

- **Decifratura**

```
1.  cipher.init(Cipher.DECRYPT_MODE, key, ivparams);
2.  input = new FileInputStream(ciphertextName);
3.  CipherInputStream decipherInput = new CipherInputStream(input, cipher);
4.  output = new FileOutputStream(decryptedtextName);
5.  r = 0;
6.  while ((r = decipherInput.read()) != -1)
7.        output.write(r);
8.  decipherInput.close();
9.  input.close();
10. output.close();
```


Gli oggetti sigillati

- La classe **SealedObject** permette di creare un oggetto e proteggerne la confidenzialità con un algoritmo crittografico
- Dato un oggetto serializzabile (**Serializable**), è possibile creare un oggetto sigillato (**SealedObject**) che incapsula l'oggetto originale, nel suo formato serializzabile, e lo sigilla (cifra).
- Il contenuto dell' oggetto sigillato può essere successivamente decifrato e de-serializzato, ottenendo così l'oggetto originale.

Gli oggetti sigillati

- Prima di essere applicato ad un oggetto sigillato, un cifrario deve essere totalmente inizializzato.
- L' oggetto originale che è stato sigillato può essere recuperato in due modi:
  - usando la **getObject** che prende un oggetto **Cipher**.
    - Chi toglie il sigillo non deve conoscere la chiave.
  - usando **getObject** che prende un oggetto **Key**.
    - **getObject** crea l'oggetto **Cipher** e lo inizializza con la chiave.
    - I parametri dell'algoritmo (**AlgorithmParameters**) sono memorizzati nell'oggetto sigillato.
    - Chi toglie il sigillo non deve tenere traccia degli algoritmi (ad esempio, IV).

Oggetti sigillati: esempio

Creazione e cifratura

```
1. String creditCard = "1234567890";
2. // Generazione di una chiave 3DES-EDE
3. KeyGenerator keyGenerator = KeyGenerator.getInstance("DESede");
4. SecretKey key = keyGenerator.generateKey();
5. // Generazione di un cipher 3DES-EDE
6. Cipher cipher = Cipher.getInstance("DESede");
7. cipher.init(Cipher.ENCRYPT_MODE, key);
8. // Creazione  e cifratura di un Sealed Object
9. SealedObject so = new SealedObject(creditCard, cipher);
```

Oggetti sigillati: esempio

- **Decifratura di un sealed object**
  1. String decryptedString = (String)so.getObject(key);
- **soluzione alternativa**
  1. cipher.init(Cipher.DECRYPT_MODE, key);
  2. String decryptedString = (String)so.getObject(cipher);

Cifratura basata su password

- Nella cifratura basata su password (*password based encryption*, *PBE*), la chiave di crittografia è derivata da una password.

$$Key \leftarrow h(Password)$$

  - **Vantaggio**. Evita di dover memorizzare le chiavi; la password è tenuta a memoria da un utente.
  - **Svantaggio**. La dimensione del **password space** è generalmente molto minore della dimensione del **key space** e quindi semplifica un brute force attack.
  - **Svantaggio**. Le password sono soggetto a dictionary attack.
- Una password deve essere semplice da ricordare ma difficile da indovinare

Password

- Generatori di password
  - **http://www.pctools.com/guides/password/**
  - generano pwd praticamente impossibili da ricordare
- Password che *appaiono* casuali ma che sono facili da ricordare
  - Bianei7na! (Biancaneve ed i sette nani)
  - GRFM0c0g (Gianluca, Roberta, Federico, Marta, zero cani, zero gatti)
- Valutazione della qualità di una password (condizione necessaria)
  - **http:www.securitystats.com/tools/password.php**

Cifratura basata su password

- Per rendere PBE più sicura si usano due tecniche
  - **Salting**. Si aggiunge alla password una stringa casuale (salt)

    $$key \leftarrow h(password, salt)$$

    ed il salt viene memorizzato con il testo cifrato

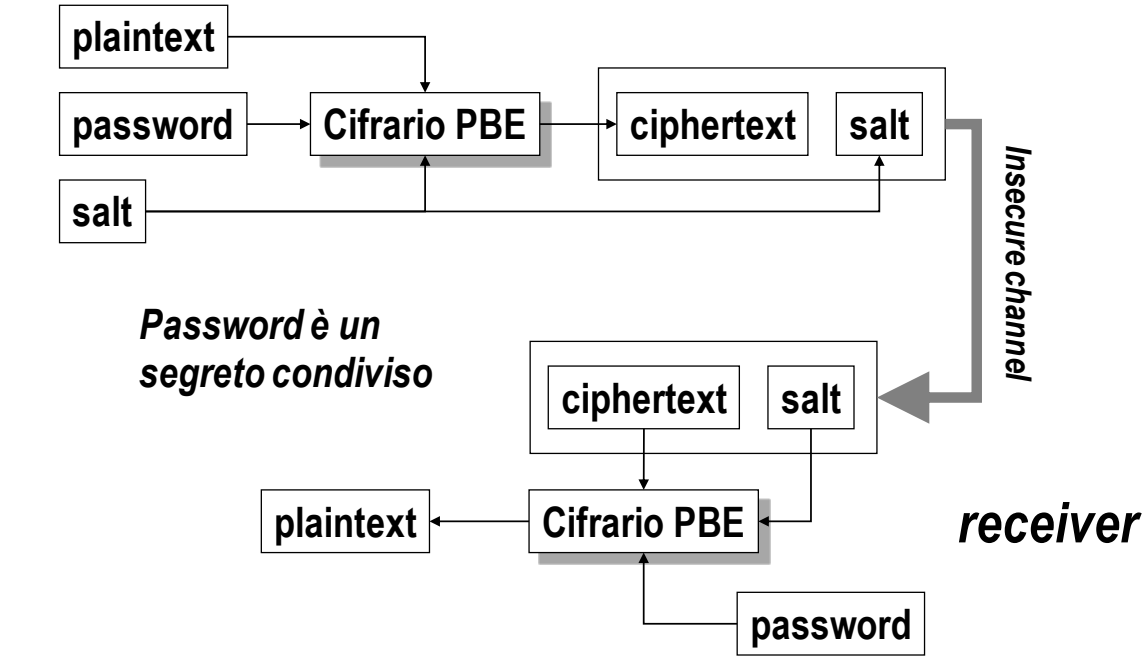
    Un nuovo salt viene generato per ogni cifratura

    In questo modo si rende impossibile la costruzione di un dizionario e per ogni tentativo l'avversario deve eseguire un hash.
  - **Repeated counting**. Tenta di aumentare il tempo necessario ad un avversario a provare una password.

    $$key \leftarrow h^r(password, salt)$$

    dove h^r consiste nella applicazione di h per r volte.

PBE: schema di comunicazione



PBE: principali classi

- Principali classi
  - **javax.crypto.spec.PBEKeySpec**
    - Factory per la creazione di una chiave basata su password
  - **javax.crypto.spec.PBEParametersSpec**
    - Permette di specificare il salt ed il numero di ripetizioni
  - **javax.crypto.SecretKeyFactory**
    - Factory per la creazione di una chiave a partire da un **PBEKeySpec**.
- Principali algoritmi
  - **PBEWithMD5AndDES, PBEWithSHAAndBlowfish, PBEWithSHAAnd128BitRC4, PBEWithSHAAndIdea-CBC, PBEWithSHAAnd3-KeyTripleDES-CBC**

PBE: passi

- **Creazione del salt di 64 bit**
  - byte[] salt = new byte[8];
  - Random random = new Random();
  - random.nextBytes(salt);
- **Creazione di un PBEKeySpec**
  - String password = “Apriti sesamo”;
  - char[] charPassword = password.toCharArray();
  - PBEKeySpec keySpec = new PBEKeySPec(charPassword);
- **Creazione di un SecretKeyFactory**
  - SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(“PBEWithMD5AndDES”);

PBE: passi

- **Creazione di una SecretKey**
  - SecretKey passwordKey = keyFactory.getInstance(keySpec);
- **Creazione di un ParameterSpec per salt e repetitions**
  - PBEParameterSpec paramSpec = new PBEParameterSpec(salt, REPETITIONS);
- Creazione ed inizializzazione di un cifrario
  - Cipher cipher = Cipher.getInstance(“PBEWithMD5AndDES”);
  - cipher.init(Cipher.ENCRYPT_MODE, passwordKey, paramSpec);

Codifica e decodifica di una chiave

- ***Problema***: una chiave di cifratura (**myKey**) deve essere memorizzata su di un supporto insicuro (ad esempio, il file system)
  - ***Gli esempi sono basati su* cipher *cifrario simmetrico ma le considerazioni valgono anche per un cifrario asimmetrico***
- Soluzione con incapsulamento (**wrap**, **unwrap**)
  - cipher.init(Cipher.WRAP_MODE, passwordKey, paramSpec);
  - byte[] encryptedKeyBytes = cipher.wrap(myKey);
  - ...
  - cipher.init(Cipher.UNWRAP_MODE, passwordKey, paramSpec);
  - Key key = cipher.unwrap(encryptedKeyBytes, “DES”, Cipher.SECRET_KEY);

Codifica e decodifica di una chiave

- **Soluzione senza incapsulamento**
  - **Cifratura**.
    - **byte[] keyBytes = myKey.getEncoded();**
    - **cipher.init(Cipher.ENCRYPT_MODE, passwordKey, paramSpec);**
    - **byte[] encryptedKeyBytes = cipher.doFinal(keyBytes);**
  - **Decifratura**
    - si usa **SecretKeySpec** che implementa **SecretKey** e può essere creato anche da un array di byte.
      - **cipher.init(Cipher.DECRYPT_MODE, passwordKey, paramSpec);**
      - **byte[] keyBytes = doFinal(encryptedKeyBytes);**
      - **SecretKeySpec myKey = new SecretKeySpec(keyBytes);**

Crittografia asimmetrica

- Cifratura asimmetrica
  - modalità e padding
  - classi ed interfacce
- Cifratura a chiave di sessione
- Memorizzazione di una chiave pubblica/privata

Cifratura asimmetrica

- ***Tipicamente utilizzati in modalità ECB***
  - Generalmente si cifrano dati di "piccole" dimensioni (in bit)
  - Per dati di "grandi" dimensioni si utilizza la *cifratura asimmetrica a chiave di sessione*
- ***Padding (per RSA)***
  - **PKCS#1**
  - **Optimal Asymmetric Encryption Padding (OAEP)**
  - http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1.index.html

In generale, ogni cifrario asimmetrico ha il proprio algoritmo di padding

Classi ed interfacce

- java.security.KeyPair
  - getPublic(), getPrivate()
- java.security.PublicKey (interface)
  - java.security.interfaces.RSAPublicKey
- java.security.PrivateKey (interface)
  - java.security.interfaces.RSAPrivateKey
  - java.security.interfaces.RSAPrivateCrtKey
- java.security.KeyPairGenerator
  - genKeyPair()

Cifratura con RSA

- Creazione del cifrario
  - Cipher cipher = Cipher.getInstance("RSA/NONE/NoPadding", "BC");
- Creazione delle chiavi
  - KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA", "BC");
  - generator.initialize(1024);
  - KeyPair pair = generator.generateKeyPair();
  - Key pubKey = pair.getPublic();
  - Key privKey = pair.getPrivate();

Cifratura con RSA

- **Cifratura**
  - **cipher.init(Cipher.ENCRYPT_MODE, pubKey);**
  - **byte[] cipherText = cipher.doFinal(input);**
- **Decifratura**
  - **cipher.init(Cipher.DECRYPT_MODE, privKey);**
  - **byte[] plainText = cipher.doFinal(cipherText);**

Cifratura a chiave di sessione

- **Problema**.
  - La cifratura asimmetrica è circa 1000 volte più lenta di quella simmetrica e quindi non è adatta a cifrare grandi quantità di dati
- **Soluzione**
  - Let **(e, d)** a public-private key pair
  - Let **P** a "large" payload
  - **K** $\leftarrow$ ***newSymmetricKey*()**
  - **M = [E_k(P), E_e(K)]**

Cifratura a chiave di sessione

- **Vantaggi**
  - Il processo di cifratura è complessivamente più veloce
  - Il sistema è complessivamente più sicuro
    - Con la chiave pubblica sono cifrate piccole quantità
      - meno materiale per la crittoanalisi
      - la chiave pubblica può durare di più
    - Con la chiave simmetrica sono cifrate grandi quantità di dati ma è rinnovata per ogni cifratura

Codifica e decodifica di chiavi
(RSA)

- *Chiave asimmetrica (struttura complessa)*
  - codifica: **getEncoded**
    - chiave pubblica: X.509
    - chiave privata: PKCS#8
  - decodifica
    - public key: **KeyFactory, X509EncodedKeySpec**
      - **X509EncodedKeySpec keySpec = new X509EncodedKeySpec(keyBytes);**
      - **KeyFactory keyFactory = KeyFactory.getInstance("RSA");**
      - **PublicKey publicKey = keyFactory.generatePublic(keySpec);**
    - private key: **KeyFactory, PKCS8EncodedKeySpec**
      - **PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(keyBytes);**
      - **KeyFactory keyFactory = KeyFactory.getInstance("RSA");**
      - **PublicKey publicKey = keyFactory.generatePublic(keySpec);**

Key agreement

- Classi principali
- Esempio con **Diffie-Hellman**

Classi principali

- **javax.crypto.KeyAgreement**
  - **getInstance()**: crea un oggetto *key agreement*
  - **init()**: inizializza questo oggetto *key agreement*
  - **doPhase()**: si passano le chiavi pubbliche degli altri *peer* a questo oggetto *key agreement*
  - **generateSecret()**: quando tutte le chiavi sono state passate, si genera il segreto condiviso

Esempio: Diffie-Hellman

- Definire i parametri ***p*** e ***g***
  - *Simple Key management for Internet Protocols* (SKIP)
  - **p** ← SKIP modulus (512, 1024, 2048)

```
private static final byte SKIP_1024_MODULUS_BYTES[] = {
    (byte)0xF4, (byte)0x88, (byte)0xFD, (byte)0x58,  (byte)0x4E, (byte)0x49, (byte)0xDB, (byte)0xCD, (byte)0x20, (byte)0xB4, (byte)0x9D, (byte)0xE4,
    (byte)0x91, (byte)0x07, (byte)0x36, (byte)0x6B,  (byte)0x33, (byte)0x6C, (byte)0x38, (byte)0x0D,  (byte)0x45, (byte)0x1D, (byte)0x0F, (byte)0x7C,
    (byte)0x88, (byte)0xB3, (byte)0x1C, (byte)0x7C,  (byte)0x5B, (byte)0x2D, (byte)0x8E, (byte)0xF6,  (byte)0xF3, (byte)0xC9, (byte)0x23, (byte)0xC0,
    (byte)0x43, (byte)0xF0, (byte)0xA5, (byte)0x5B,  (byte)0x18, (byte)0x8D, (byte)0x8E, (byte)0xBB,  (byte)0x55, (byte)0x8C, (byte)0xB8, (byte)0x5D,
    (byte)0x38, (byte)0xD3, (byte)0x34, (byte)0xFD,  (byte)0x7C, (byte)0x17, (byte)0x57, (byte)0x43,  (byte)0xA3, (byte)0x1D, (byte)0x18, (byte)0x6C,
    (byte)0xDE, (byte)0x33, (byte)0x21, (byte)0x2C,  (byte)0xB5, (byte)0x2A, (byte)0xFF, (byte)0x3C,  (byte)0xE1, (byte)0xB1, (byte)0x29, (byte)0x40,
    (byte)0x18, (byte)0x11, (byte)0x8D, (byte)0x7C,  (byte)0x84, (byte)0xA7, (byte)0x0A, (byte)0x72,  (byte)0xD6, (byte)0x86, (byte)0xC4, (byte)0x03,
    (byte)0x19, (byte)0xC8, (byte)0x07, (byte)0x29,  (byte)0x7A, (byte)0xCA, (byte)0x95, (byte)0x0C,  (byte)0xD9, (byte)0x96, (byte)0x9F, (byte)0xAB,
    (byte)0xD0, (byte)0x0A, (byte)0x50, (byte)0x9B,  (byte)0x02, (byte)0x46, (byte)0xD3, (byte)0x08,  (byte)0x3D, (byte)0x66, (byte)0xA4, (byte)0x5D,
    (byte)0x41, (byte)0x9F, (byte)0x9C, (byte)0x7C,  (byte)0xBD, (byte)0x89, (byte)0x4B, (byte)0x22,  (byte)0x19, (byte)0x26, (byte)0xBA, (byte)0xAB,
    (byte)0xA2, (byte)0x5E, (byte)0xC3, (byte)0x55,  (byte)0xE9, (byte)0x2F, (byte)0x78, (byte)0xC7
};
```


Esempio: Diffie-Hellman

- ***g*** ← 2

- Rappresentare i parametri ***p*** e ***g*** come **java.math.BigInteger**
  1. **private static final BigInteger P = new BigInteger(SKIP_1024_MODULUS_BYTES);**
  2. **private static final BigInteger G = BigInteger.valueOf(2);**
- Inserimento di **p** e **g** in un **DHParameterSpec**
  1. **private static final DHParameterSpec PARAMETER_SPEC = new DHParameterSpec(P,G);**
- Generazione di una coppia di chiavi pubblica e privata
  1. **KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");**
  2. **kpg.initialize(PARAMETER_SPEC);**
  3. **KeyPair myKeyPair = kpg.genKeyPair();**

Esempio: Diffie-Hellman

- Recupero della chiave pubblica del peer da uno stream (file, network) in formato **byte[] peerKeyBytes**
  1. **KeyFactory kf = KeyFactory.getInstance("DH");**
  2. **X509EncodedKeySpec x509Spec = new X509EncodedKeySpec(peerKeyBytes)**
  3. **PublicKey peerPublicKey = kf.generatePublic(x509Spec);**
- Esecuzione del key agreement
  1. **KeyAgreement ka = KeyAgreement.getInstance("DH");**
  2. **ka.init(myKeyPair.getPrivate);**
  3. **ka.doPhase(peerPublicKey);**
- Generazione del segreto condivisto
  1. **byte[] sharedSecret = ka.generateSecret();**

Message digest e MAC

- Message digest
  - Le classi principali
  - Esempio
  - Message digest ed i flussi
- MAC
  - HMAC
  - Le classi principali

Message Digest: classi

- **java.security.MessageDigest**
  - **getinstance()** genera un oggetto **MessageDigest**
  - **update()** applica l'algoritmo di message digest
  - **digest()** ritorna il digest

Message digest: esempio

- Digest di una stringa
  - **String text = "Nel mezzo del cammin di nostra vita mi trovai in una selva oscura";**
  - **byte[] textbyte = text.getBytes();**
- Ottenere un oggetto **MessageDigest**
  - **MessageDigest md = MessageDigest.getInstance("MD5");**
- Applicazione dell'algoritmo di *message digest*
  - **md.update(textbyte);**
- Ritorno del digest
  - **byte[] hash = md.digest();**

Message digest e flussi

- Il package **java.security** contiene le classi **DigestInputStream** e **DigestOutputStream** che permettono di aggiornare (update) il diget mentre i dati vengono letti e scritti, rispettivamente.
- Esempio
  - **MessageDigest md = MessageDigest("SHA");**
  - **DigestInputtream digestIn = new DigestInputStream(*existing input stream*, md);**
  - **while(digestIn.read() != -1);**
  - **byte[] theDigest = md.digest();**

MAC: classi

- javax.crypto.Mac
  - getInstance
  - init
  - update
  - doFinal

MAC: esempio

- Creazione diun oggetto MAC
  - **Mac mac = Mac.getInstance(“HmacSHA1”);**
- Inizializzazione del MAC
  - **mac.init(key); // key è una chiave per HmacSHA1**
- Applicazione dell’algoritmo
  - **mac.update(plainText); // byte[] plainText**
- Restituzione del MAC
  - **byte[] result = mac.doFinal();**

Digital signatures

- Le principali classi per la firma digitale
- I certificati digitali
  - Note sullo standard X.509v3
  - Classi principali
  - Keystore
  - keytool

Le classi principali

- **java.security.Signature**
  - **getInstance()**: ritorna un oggetto per la creazione e la verifica di firme digitali
  - **initSign(), initVerify()**: inizializza l'oggetto per la firma o la verifica
  - **update()**: si trasmettono i dati da firmare o verificare
  - **sign()**: si firmano i dati trasmessi
  - **verify()**: si verificano i dati trasmessi

Esempio: firma e verifica

- Generazione di una coppia di chiavi
  1. **KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA", "BC");**
  2. **kpg.initialize(1024);**
  3. **KeyPair keyPair = kpg.generateKeyPair();**
- Generazione di un signature engine
  4. **Signature sig = Signature.*getInstance*("MD5WithRSA");**

Generazione della firma digitale

- Prendere il testo da firmare
  5. **String text = "I am a liar";**
  6. **byte[] data = text.getBytes();**

(*continua*)

Esempio: firma e verifica

- Inizializzazione dell'engine per la firma
  5. **sig.initSign(keyPair.getPrivate());**
- Trasmissione dati all'engine
  6. **sig.update(data);**
- Generazione della firma digitale
  7. **byte[] signatureBytes = sig.sign();**

(*continua*)

Esempio: firma e verifica

Verifica della firma digitale

- Inizializzazione del signature engine per la verifica
  1. **sig.initVerify(keyPair.getPublic());**
- Trasmissione dati all'engine
  2. **sig.update(data);**
- Verifica della firma

```
3. try {
4.     verified = sig.verify(signatureBytes);
5. } catch(SignatureException  e) {
6.     verified = false;
7. }
```

Certificati

- X.509
  - http://www.ietf.org/rfc2459.txt
  - X.509v1 comprende:
    - versione
    - numero di serie
    - algoritmo di firma
    - validità
    - soggetto (X.500)
    - distributore (X.500)
    - chiave pubblica del soggetto
    - firma
  - X.509v2
    - Issuer Unique Identifier
    - Subject Unique Identifier
  - X.509v3 (estensioni)
    - Attributi del soggetto e del distributore
    - Utilizzo e comportamenti della chiave
    - Limitazione del percorso di certificazione

Nomi nel formato X.500

ATTRIBUTO	CONTENUTI
CN	Common Name
OU	Organization Unit
O	Organization
L	Location
ST	State
C	Country

- Esempi di nomi X.500
  - Server
    - CN = goblin.adm.unipi.it
    - O = University of Pisa
    - ST = Italy
    - C = IT
  - Certification Authority
    - CN = Visa eCommerce Root
    - OU = Visa International Service Association
    - O = VISA
    - C = US

Classi principali

- **java.security.cert.Certificate** è una classe astratta che incapsula un certificato
  - **getPublicKey**
  - **verify**

I certificati sono immutabili

- **java.security.cert.X509Certificate** è una classe astratta che incapsula un certificato X.509
- **java.security.cert.CertificateFactory**
  - **getInstance**

Keystore

- ***Keystore*** è un repository di *chiavi private* e *certificati affidabili*
  - Il repository può essere implementato con un file, un database, un server LDAP
  - **java.security.KeyStore**
    - L'operazione **getInstance** genera un keystore
- Ogni chiave/certificato è associato ad un nome (*alias*)
    - l'operazione **aliases** ritorna l'elenco di tutti gli alias
- Keystore predefinito: **.keystore**
  - Windows: **C:\Documents and Settings*user name*\.keystore**
  - Linux: **/home/*user name*/.keystore**

Keystore

- Un *certificato affidabile* è un certificato che si *assume* appartenere al soggetto specificato nel certificato
  - Esempio: il certificato di una Certification Authority
  - L'operazione **setCertificateEntry** permette di aggiungere un certificato affidabile al keystore
- Le *chiavi private* da utilizzare per le firme digitali
  - La chiave deve essere associata ad un certificato (non affidabile)
  - L'operazione **setKeyEntry** permette di aggiungere una chiave al keystore

Keytool

- Permette di creare certificati
  - **-keystore**
  - **-certreq**
  - **-delete**
  - **-export**
  - **-genkey**
  - **-help**
  - **-identiypub**
  - **-import**
  - **-keyclone**
  - **-keypasswd**
  - **-list**
  - **-printcert**
  - **-selfcert**
  - **-storepasswd**

Creazione di un certificato

- Creazione di un certificato
  - **keytool -genkey -alias *name***
    - **-genkey**. Genera una coppia di chiavi ed un certificato ***auto-firmato***.
    - È possibile specificare l'algorithmo con **–keyalg**
- Elenco del contenuto del keystore
  - **keytool –v -list**
- Esportazione di un certificato
  - **keytool –export –alias *name* –file *filename***

Esempio: lettura di un certificato da file

- Creazione di un Certification Factory
  - **CertificateFactory certFact = CertificateFactory.*getInstance*("X.509");**
- Apertura del file che contiene il certificato (preventivamente esportato)
  - **FileInputStream fis = new FileInputStream(*file name*);**
- Creazione di un certificato a partire dal contenuto del file
  - **Certificate cert = certFact.generateCertificate(fis);**
  - **fis.close();**
  - **System.*out*.println(cert);**

Esempio: output

```
[
[
  Version: V3
  Subject: CN=Gianluca Dini, OU=Dept. of Information Engineering, O=University of
   Pisa, L=Pisa, ST=Pisa, C=IT
  Signature Algorithm: SHA1withDSA, OID = 1.2.840.10040.4.3

  Key:  Sun DSA Public Key
    Parameters:DSA
p:     fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
       455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
       6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
       83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
q:     9760508f 15230bcc b292b982 a2eb840b f0581cf5
g:     f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
       5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
       3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
       cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a
y:     8a1c9ca7 73b8594f 4899e1da 49b63e90 90220123 dd3795b9 b481f964 9bf69e83
       7a4ed70a 52146ba5 4ca145e9 3bfd4f92 af4577a9 641c9a7b 4775efdd fb9cf375
       a7b5619a 333400ba bb4ea185 bdecbcde 14a63b74 e2edc574 3b41749c bde0d882
       5d4e1237 35b370f3 365f5d40 78288179 9da4c4c1 71131104 09d02e07 e871052d
```


Esempio: output

```
  Validity: [From: Sat Apr 26 16:10:57 CEST 2008,
              To: Fri Jul 25 16:10:57 CEST 2008]
  Issuer: CN=Gianluca Dini, OU=Dept. of Information Engineering, O=University of
   Pisa, L=Pisa, ST=Pisa, C=IT
  SerialNumber: [    481337f1]

]
  Algorithm: [SHA1withDSA]
  Signature:
0000: 30 2C 02 14 1D 80 03 C0   1F 38 33 41 5D E1 C5 E3  0,.......83A]...
0010: D3 BD 18 73 42 3A F6 6B   02 14 54 C5 2E F0 58 B7  ...sB:.k..T...X.
0020: AB DB CA B3 1F 2A E8 95   B3 7C C4 38 EA 70        .....*.....8.p

]
```

Lettura da keystore

- Ricavare il *file name* del keystore
  - String username = System.getProperty("user.home");
  - String keystoreFilename = username + "/.keystore";
- Determinare *alias* e *password*
  - char[] password = args[1].toCharArray();
  - String alias = args[0];
- Ottenere un *KeyStore engine*
  - KeyStore keystore = KeyStore.getInstance("JKS");

Lettura da keystore

- Caricare il *keystore* nel KeyStore engine
  - FileInputStream fis = new FileInputStream(keystoreFilename);
  - keystore.load(fis, password);
- Leggere il certificato dal keystore
  - Certificate cert = keystore.getCertificate(alias);
  - System.out.println(cert);

CRL

- **java.security.cert.X509CRL**
  - ***scaricare un CRL in un file*** **(`http://crl.verisign.com`);**
  - **FileInputStream fis = new FileInputStream("crl_filename");**
  - **CertificateFactory cf = CertificateFactory.getInstance("X.509");**
  - **X509CRL crl = (X509CRL)cd.generateCRL(fis);**
  - **fis.close();**
  - **…**
  - **if (crl.isRevoked())**
  - **System.out.println("Il certificato e' stato revocato.");**
  - **else**
  - **System.out.println("Il certificato e' ok.");**

SSL

- **Java Security Socket Extension (JSSE)**
  - JSSE definisce un API per usare SSL in Java
  - JSSE fornisce una implementazione di riferimento della API
  - Provider
    - **http://java.sun.com/security**
    - **jcert.jar, jnet.jar, jsse.jar**
    - **com.sun.net.ssl.internal.ssl.Provider**

Socket SSL

- ***Classe* javax.net.ssl.SSLSockets**
  - sottoclasse di **java.net.Socket**
- ***Server***
  1. **SSLServerSocketFactory ssf = (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();**
  2. **ServerSocket ss = ssf.createServerSocket(PORT);**
- ***Client***
  1. **SSLSocketFactory sf = (SSLSocketFactory)SSLSocketFactory.getDefault();**
  2. **Socket s = sf.createSocket(HOST, PORT);**

Socket SSL

- ***Gestione dei certificati***
  - Fornire al server una coppia di chiavi ed un certificato autofirmato
    - **keytool –genKey –alias *serverAlias* –v –keyAlg RSA –keystore .serverKeystore**
  - Fornire al client un certificato del server
    - **keytool –export –alias *serverAlias* –file serverCert.cer**
    - **keytool –import –alias serverAlias –file serverCert.cer –keystore .clientKeystore**
    - Specificare che il certificato è fidato
      - Trusted certification authority per impostazioni predefinita stanno nel keystore **$JRE_HOME/lib/security/cacerts**
      - Un trusted key store può essere specificato per mezzo della proprietà **javax.net.ssl.trustStore**
        (vedi esempio prox. pag.)

Socket SSL

- ***Esecuzione***
  - Server
    - **java –Djavax.net.ssl.keyStore=.serverKeystore –Djavax.net.ssl.keyStorePassword=password Server**
  - Client
    - **java –Djavax.net.ssl.trustStore=.clientKeystore Client**

Uso di HTTPS

- **Classi: java.net.URL**
- **Server**
  - *come nell'esempio precedente*
- **Client**
  - *Codice*
    - **URL url = new URL("https://www.verisign.com/")**
    - **BufferedReader in = new BufferedReader(url.openStream());**
    - ***lettura orientata alla linea***
  - *Impostare il gestore di URL in modo che possa trovare le classi SSL*
    - impostare la proprietà **java.protocol.handler.pkgs = com.sun.net.ssl**
      - all'avvio: **-D**
      - a programma: **System.setProperty(*name*, *value*)**