# Java Cryptography Architecture

# Crittografia con JAVA

- Si utilizzano due API
- Java Cryptographic Architecture (JCA)
  - Fa parte del run-time environment di Java 2
    - **java.security.***
- Java Cryptographic Extensions (JCE)
  - Estende l'API di JCA API ed è integrata nell'SDK di Java 2 a partire dalla release 1.4
    - **javax.crypto.***

# JCA vs JCE

- JCA (javax.security.*)
  - Definisce le classi astratte per gran parte delle funzionalità crittografiche, ma ne implementa solo alcune spefiche
  - Si occupa della gestione delle chiavi e dei certificati digitali, di firme digitali, …
  - Contiene classi che **non sono** soggette a limitazioni sull'esportazione
- JCE (javax.crypto.*)
  - Definisce API complete e implementa tutte la altre funzionalità
  - Fornisce implementazioni di primitive crittografiche: cifratura, hashing, …
  - Contiene classi che **sono** soggette a limitazioni sull'esportazione

# Caratteristiche JCA/JCE

- Indipendenza
  - Le applicazioni non devono implementare servizi di sicurezza, ma li possono richiedere alla piattaforma tramite un'interfaccia standard
- Interoperabilità
  - I servizi crittografici sono forniti da provider che possono essere scelti in un insieme di provider disponibili
    - L'applicazione non è vincolata al provider e il provider non è vincolato all'applicazione
- Estendibilità
  - È possibile sviluppare/aggiungere provider seguendo una specifica prefissata

# Provider

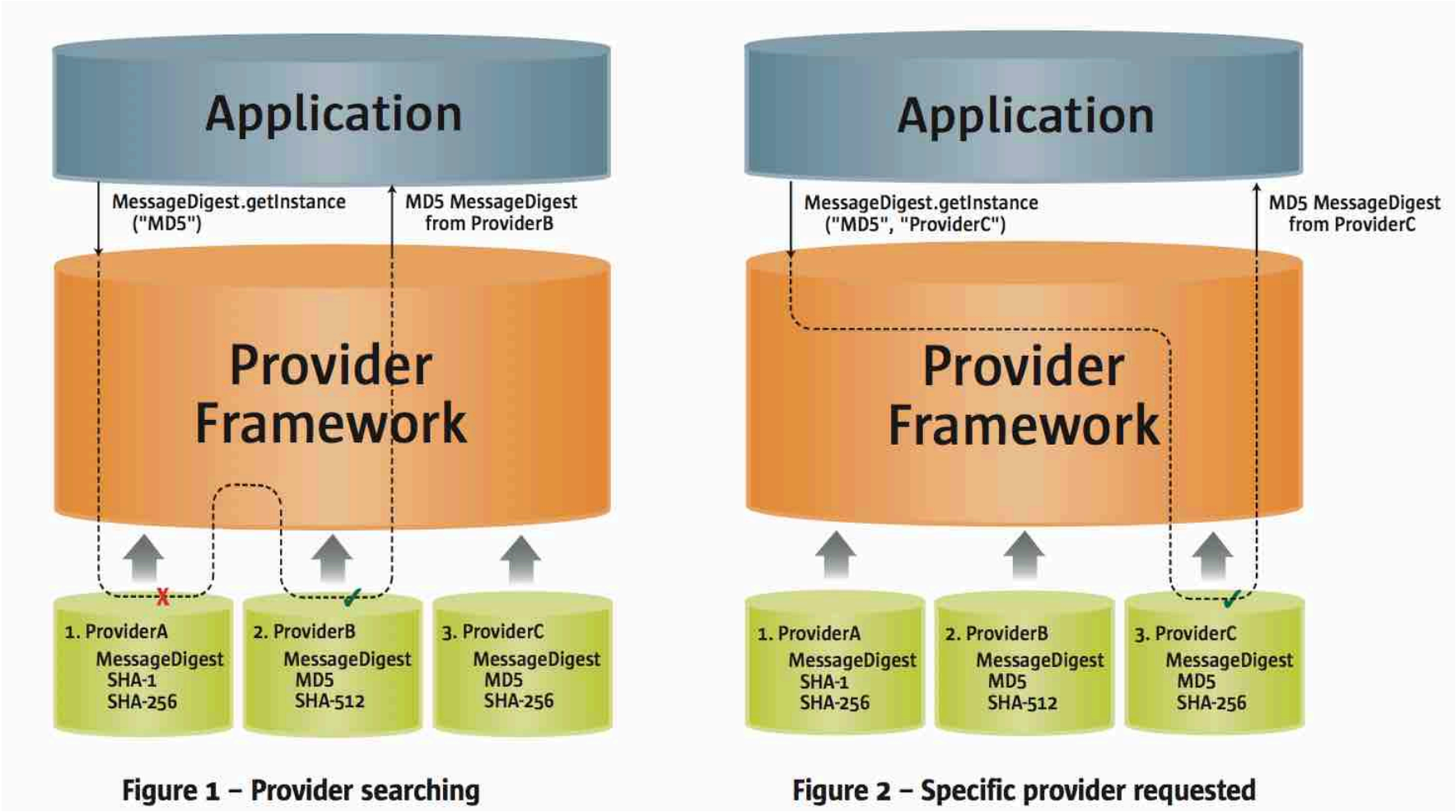



**Figure 1 – Provider searching**

**Figure 2 – Specific provider requested**

# Come scegliere il provider

- Indicazione del provider implicita
  - md = MessageDigest.getInstance("MD5")
- Indicazione del provider esplicita
  - md = MessageDigest.getInstance("MD5","ProviderC")

# Ordine dei provider

- I provider sono localizzati in
  $JAVA_HOME/jre/lib/ext
- L'ordine di preferenza dei provider è indicato nel file java.security localizzato in $JAVA_HOME/jre/lib/security

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
……
```

# Altri provider

- IAIK/JCE
  - http://jce.iaik.tugraz.at/
- Bouncy Castle
  - http://www.bouncycastle.org/
- Spongy Castle
  - Versione ridotta di Bouncy Castle per Android
  - https://rtyley.github.io/spongycastle/

# Installazione dinamica di un provider

- Realizzata tramite due classi definite in java.security
- Provider
  - Classe che è usata per ottenere informazioni sui provider installati
- Security
  - Classe che è usata per aggiungere, rimuovere e modificare i provider

# Due tipologie di classi

- Tipo engine
  - Classe astratta che dichiara le funzionalità di una primitiva (algoritmo) crittografica
- Tipo provider
  - Classe che implementa un certo insieme di funzionalità crittografiche per un provider

# Principali classi engine di java.security

- **Key**
  - Definisce le funzionalità di una chiave opaca
- **KeySpec**
  - Definisce una chiave di tipo trasparente
- **KeyFactory**
  - Converte una chiave da opaca a trasparente
- **KeyPairGenerator**
  - Genera una coppia di chiavi asimmetriche

Crittografia asimmetrica

# Opaca vs Trasparente

- **Opaca**
  - Non si può accedere a tutte le *parti* che costituiscono una chiave, si possono ottenere solo informazioni limitate tramite i metodi
    - getAlgorithm, getFormat, e getEncoded
- **Trasparente**
  - Accesso completo alla chiave tramite metodi del tipo getXXX, Ad esempio per una chiave RSA
    - getPublicExponent() e getPrivateExponent()

# Ancora su chiavi trasparenti

- Una rappresentazione trasparente delle chiavi significa che si può accedere ad ogni valore della chiave singolarmente tramite uno dei metodi get definiti nella corrispondete classe di specifica.

**FIrma digitale**

- Ad esempio, **DSAPrivateKeySpec** definisce i metodi getX, getP, getQ e getG, per accedere alla chiave privata x e ai parametri dell'algoritmo DSA usati per calcolare la chiave (i primi p e q e la base g)

# Principali classi engine di java.security

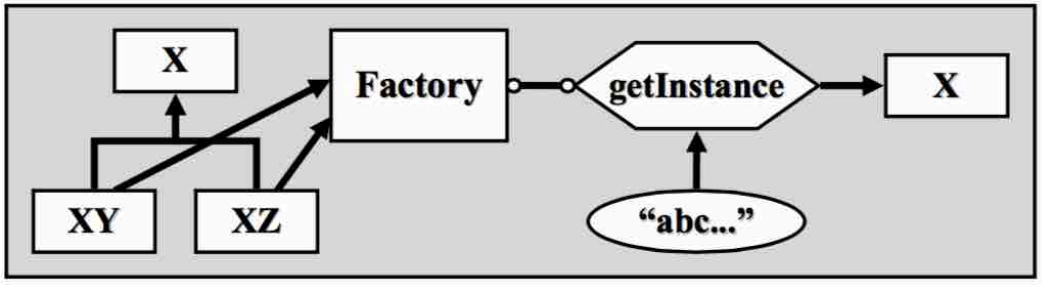
- **AlgorithmParameters**
  - Gestisce i parametri di un algoritmo crittografico
- **AlgorithmParameterGenerator**
  - Genera i set di parametri di un algoritmo crittografico
- **MessageDigest**
  - Calcola l'hash (message digest) di un messaggio
- **SecureRandom**
  - Genera numeri  pseudo-casuali crittograficamente forti

# Principali classi engine di javax.crypto

- **Cipher**
  - Offre funzionalità di cifratura e decifrazione di dati mediante uno specifico algoritmo
- **Mac**
  - Offre funzionalità di Message Authentication Code (MAC)
- **CipherInputStream** / **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

# Factory Pattern

- Tutte le classi di tipo engine della JCA e JCE sono adoperate in maniera simile tramite un Factory Pattern




- Per istanziare un oggetto
  - Non si usa la parola chiave new
  - Si usa un metodo statico getIstance(String nomeIstanza)

# Factory Pattern

- È un design pattern che si basa sui factory method
  - Per convenzione i factory method si chiamano **getInstance()**
- Un factory method è un tipo speciale di metodo statico che restituisce un'istanza di una classe

# Esempi

A provider differenti potrebbero corrispondere nomi differenti

**KeyGenerator** kg = KeyGenerator.getInstance("TripleDES");

**KeyPairGenerator** kpg = KeyPairGenerator.getInstance("RSA");

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

**CertificateFactory** cf = CertificateFactory.getInstance("X.509");

# Generazione di chiavi – cifrari a blocchi

Istanzia un generatore di chiavi di tipo DES

```
KeyGenerator keyGenerator =
        KeyGenerator.getInstance("DES");
```

```
keyGenerator.init(56);
```

Si può specificare un generatore casuale per l'inizializzazione

Inizializza il generatore di chiavi con la lunghezza in bit della chiave (per DES 56 bit)
Altri algoritmi hanno lunghezze di chiavi variabili

```
SecretKey secretKey =
        keyGenerator.generateKey();
```

Genera una chiave DES opaca

# Che parametri passare a getInstance?

- Ci sono dei nomi standard da utilizzare
  - Sono mnemonici
  - Dipendono dal provider utilizzato
- Per i provider installati di default si può far riferimento a **Java Security Standard Algorithm Names**

  http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html

  http://download.java.net/java/jdk9/docs/specs/security/standard-names.html

# Parametri di KeyGenerator.getInstance

- AES
- ARCFOUR (Cifrario simile a RC4)
- Blowfish
- DES
- DESede (triple DES)

# SecureRandom

- Fornisce un generatore pseudo-casuale crittograficamente forte

```
SecureRandom random = new SecureRandom();
byte bytes[] = new byte[8];
random.nextBytes(bytes); //genera 8 byte casuali
```

Genera una chiave DES usando il generatore indicato

```
keyGenerator.init(56, new SecureRandom());
```

# SecureRandom.getInstance()

- Nomi che si possono usare
  - NativePRNG
  - NativePRNGBlocking
  - NativePRNGNonBlocking
  - PKCS11, SHA1PRNG, Windows-PRNG

```
SecureRandom random =
                SecureRandom. getInstance("PKCS11");
byte bytes[] = new byte[8];
random.nextBytes(bytes); //genera 8 byte casuali
```

# Cifrare

String text ="Ciao a tutti!";

Otteniamo un'istanza del cifrario

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

Dobbiamo prima inizializzare secretKey

Inizializziamo il cifrario per cifrare

```
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

Convertiamo la stringa in byte[] codificandola per evitare problemi di compatibilità su macchine diverse
La codifica di default dipende dalla macchina

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

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

Cifriamo

# Combinazioni per l'istanza

- Quando si crea un'istanza del cifrario è necessario specificare tre informazioni separate da / (slash). Solo la prima informazione è obbligatoria
  - Nome dell'algoritmo
  - Modalità di cifratura (opzionale)
  - Tipo di padding (opzionale)
- Il provider specifica un valore di default per i parametri opzionali

# Alcune Combinazioni Cifrario/Modalità/Padding

- AES/CBC/NoPadding (chiave di 128 bit)
- AES/CBC/PKCS5Padding (chiave di 128 bit)
- AES/ECB/NoPadding (chiave di 128 bit)
- AES/ECB/PKCS5Padding (chiave di 128 bit)
- DES/CBC/NoPadding (chiave di 56 bit)
- DES/CBC/PKCS5Padding (chiave di 56 bit)
- DES/ECB/NoPadding (chiave di 56 bit)
- DES/ECB/PKCS5Padding (chiave di 56 bit)

# Decifrare

Inizializziamo il cifrario per decifrare

```
cipher.init(Cipher.DECRYPT_MODE, secretKey);
```

Decifriamo

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

Convertiamo da byte[] a String

```
String output = new String(decryptedText,"UTF8");
```

# Eccezioni

- Per far funzionare gli esempi nelle slide è necessario aggiungere la gestione delle opportune eccezioni
  - NoSuchAlgorithmException
  - NoSuchPaddingException
  - InvalidKeyException
  - UnsupportedEncodingException
  - BadPaddingException
  - IllegalBlockSizeException

# doFinal vs update

- Invece del metodo doFinal si può usare il metodo update
  - Esso cifra/decifra array di byte continuando l'operazione su cifrature/decifrature multiple
  - Utilizza un buffer interno, quando il buffer è pieno lo cifra/decifra e lo restituisce (senza padding)
    - I dati in eccesso vengono conservati per la prossima cifratura/decifratura
  - Se i dati input sono più *corti* della dimensione del blocco restituisce null
  - doFinal svuota il buffer e aggiunge l'eventuale padding

Il buffer è grande quanto il blockSize

# SecretKeyFactory

- Permette di convertire una chiave da trasparente (**KeySpec**) a opaca (**Key**) e viceversa

- **KeySpec** (interfaccia) chiave + metadati
  - Una specifica trasparente del *key material* che costituisce una chiave crittografica
  - Ogni primitiva crittografica ha la sua implementazione (e.g., **DESKeySpec**)

Per AES non c'è nei provider di default – in quello IBM c'è

# Esempio DES da trasparente a opaca

```
SecureRandom random = new SecureRandom();
byte desKey[] = new byte[8];
random.nextBytes(desKey);
```

Chiave trasparente

```
DESKeySpec desKeySpec new DESKeySpec(desKey);

SecretKeyFactory factory =
                    SecretKeyFactory.getInstance("DES");

SecretKey  secretKey =                          Chiave opaca
                    factory.generateSecret(desKeySpec);
```

# Attenzione

- **Cipher** usa **Key** (o **SecretKey**) per inizializzare gli algoritmi di cifratura/decifrazione

- La chiave può essere convertita in un formato opportuno per la trasmissione o la memorizzazione
  - La trasformiamo da opaca a trasparente

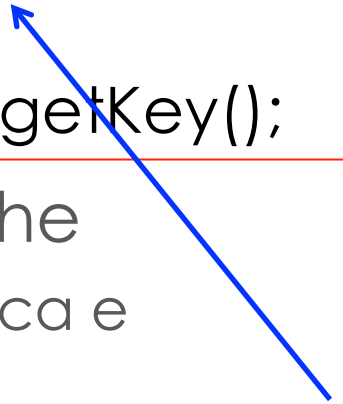
# Da opaca a trasparente

- Si usa il metodo getKeySpec di **SecretKeyFactory**
  - Per cifrari a blocchi praticamente non c'è differenza tra opaca e trasparente

```
DESKeySpec newDESKeySpec = (DESKeySpec)
        factory.getKeySpec(secretKey, DESKeySpec.class);

byte [] transparentDESKey = newDESKeySpec.getKey();
```

- Simile approccio per chiavi asimmetriche
  - Per cifrari asimmetrici c'è differenza tra opaca e trasparente



Formato da utilizzare per la conversione

# SecretKeySpec

- Permette di definire una chiave in maniera indipendente dal provider
- Può essere usata per costruire una **SecretKey** da un array di byte senza dover passare da **SecretKeyFactory**
  - Possiamo farlo per quei cifrari la cui chiave può essere rappresentata da un array di byte e non ha parametri associati ad essa

# Esempio di SecretKeySpec per AES

```
SecureRandom random =
                        new SecureRandom();

byte aesKey[] = new byte[16];

random.nextBytes(aesKey);

SecretKey secretKey =
        new SecretKeySpec(aesKey,
                          0, aesKey.length,
                           "AES");
```

# Altro modo per generare una chiave

```
KeyGenerator keyGenerator = null;
                try {
                        keyGenerator = KeyGenerator.getInstance("AES");
                } catch (NoSuchAlgorithmException e) {
                        System.out.println("Algoritmo non supportato");

                }

                //Inizializziamo il KeyGenerator
                keyGenerator.init(128, new SecureRandom());

                //Generiamo la chiave
                SecretKey secretKey = keyGenerator.generateKey();
```

# Base64

- È un formato utilizzato per codificare dati binari (insieme di ottetti) in formato ASCII
- La codifica suddivide il dato binario in gruppi da 6 bit (valori da 0 a 63) a cui associa un carattere ASCII (8 bit)
  - Ogni 24 bit si generano 4 caratteri ASCII (32 bit)
  - Se la lunghezza del messaggio binario non è un multiplo di 6 si aggiungono zeri alla fine ed eventualmente 0, 1 o 2 caratteri "**=**" (uguale)
  - Ogni carattere di padding indica l'aggiunta di una coppia di zeri

# Da byte[] a String (Base64)

```
byte byteKey[] = secretKey.getEncoded();

String encodedKey =
Base64.getEncoder().encodeToString(byteKey);
```

# Da String (Base64) a SecretKey

```
// decode the base64 encoded string

byte[] decodedKey =
Base64.getDecoder().decode(encodedKey);

// rebuild key using SecretKeySpec

SecretKey originalKey =
    new SecretKeySpec(decodedKey, 0,
                      decodedKey.length, "AES");
```

# Modalità CBC

- È sufficiente usare CBC come parametro per la modalità
- Il vettore di inizializzazione è generato automaticamente dall'istanza del cifrario
- Per ottenere il suo valore usiamo il metodo getIV
  - byte[] iv = cipher.getIV();
- Volendo, possiamo specificare il vettore di inizializzazione

# IvParameterSpec

```
SecureRandom random = new SecureRandom();
byte IVBytes[] = new byte[16]; //Per AES
random.nextBytes(IVBytes);
IvParameterSpec
                    iv = new IvParameterSpec(IVBytes);
cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);
```

# getIV()

- Metodo di **IvParameterSpec**
  - Restituisce il vettore di inizializzazione
  - Ogni volta che è invocato restituisce un nuovo vettore

- **IvParameterSpec** non ha altri metodi

# Password-Based Key-Derivation

- Tecnica utilizzata per generare una chiave a partire da un password (array di char) e un salt (array di byte) opzionale
- Basata sullo standard PKCS #5 2.0 Password-based key-derivation algorithm
- Si utilizzano
  - **SecretKeyFactory**
  - **PBEKeySpec**

# Esempio generazione chiave

La chiave generata può essere usata per cifrare o decifrare

```
char[] password
byte[] salt
```

Devono essere inizializzati in maniera opportuna

```
SecretKeyFactory factory =
  SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
```

Algoritmo per generare la chiave

```
KeySpec keySpec =
        new PBEKeySpec(password, salt, 65536, 128);
```

Specifiche della chiave

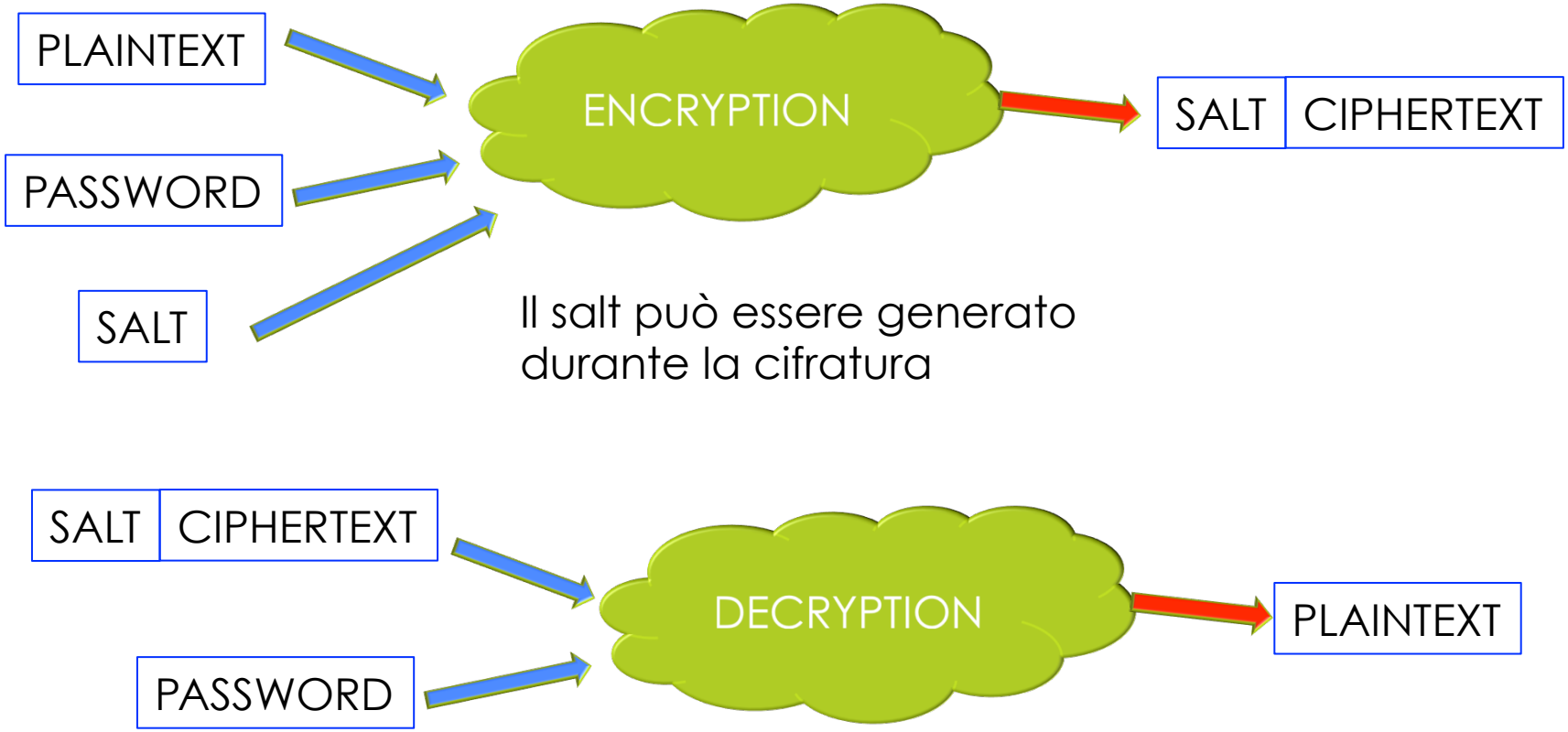
```
SecretKey tmp = factory.generateSecret(keySpec );
```

Genera una chiave generica

```
SecretKey secret =
        new SecretKeySpec(tmp.getEncoded(), "AES");
```

Genera una chiave AES

# Schematicamente



# PBKDF2WithHmacSHA256

- Algoritmo di derivazione di una chiave crittografica, definito in PKCS#5 v2.0, usando la funzione pseudo casuale HmacSHA256
  - Usa il metodo PBKDF2 della specifica PKCS#5
- Funzioni pseudo casuali
  - HmacSHA1, HmacSHA224, HmacSHA256, HmacSHA384, HmacSHA512

PKCS #5 v2.1: Password-Based Cryptography Standard
RSA Laboratories October 27, 2012

# PBEKeySpec

- PBEKeySpec(password, salt, 65536, 128)
  - password un array di caratteri
    - Deve essere tenuto segreto
  - salt un array di byte (casuale)
    - Non deve essere tenuto segreto, ma deve essere noto per decifrare
  - 65536 numero di volte che la funzione pseudo casuale deve essere applicata
  - 128 lunghezza in bit della chiave che deve essere generata

# Key Wrapping

- Cifratura/decifrazione della chiave di un cifrario
  - Utile per memorizzare la chiave su un filesystem
- Invece di scrivere del codice ad-hoc per estrarre la chiave da **Key** e poi cifrarla, possiamo usare dei metodi di **Cipher**
  - wrap con il cifrario inizializzato a WRAP_MODE
  - unwrap con il cifrario inizializzato a UNWRAP_MODE
    - I metodi cifrano e decifrano la chiave

# Esempio Key Wrapping

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
keyGenerator.init(128, new SecureRandom());
SecretKey secretKey1 = keyGenerator.generateKey();
SecretKey secretKey2 = keyGenerator.generateKey();
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
```

Potremmo derivare secretKey1 da una password e da un salt

secretKey2 sarà cifrata (wrapped) con secretKey1

```
cipher.init(Cipher.WRAP_MODE, secretKey1);
byte[] wrappedKey = cipher.wrap(secretKey2);
```

secretKey2 sarà decifrata (unwrapped) con secretKey1

```
cipher.init(Cipher.UNWRAP_MODE, secretKey1);
Key unwrappedKey = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
```

# Costanti di Cipher – 1

- DECRYPT_MODE
  - Constant used to initialize cipher to decryption mode
- ENCRYPT_MODE
  - Constant used to initialize cipher to encryption mode
- UNWRAP_MODE
  - Constant used to initialize cipher to key-unwrapping mode
- WRAP_MODE
  - Constant used to initialize cipher to key-wrapping mode

# Costanti di Cipher – 2

Cifrari asimmetrici

- PRIVATE_KEY
  - Constant used to indicate the to-be-unwrapped key is a "private key"
- PUBLIC_KEY
  - Constant used to indicate the to-be-unwrapped key is a "public key"
- SECRET_KEY
  - Constant used to indicate the to-be-unwrapped key is a "secret key”

# File Encryption

- Realizzato tramite CipherInputStream e CipherOutputStream di javax.crypto.
  - CipherInputStream = InputStream + Cipher
  - CipherOutputStream = OutputStream + Cipher
  - I metodi read/write elaborano i dati con Cipher prima di restituirli all'utente o scriverli nell'OutputStream
  - Cipher deve essere opportunamente inizializzato prima di usare CipherInputStream e/o CipherOutputStream

# Public Key Encryption

```
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");

keyPairGenerator.initialize(1024, new SecureRandom());

KeyPair userKey = keyPairGenerator.generateKeyPair();
```

```
PrivateKey userKeyPr = userKey.getPrivate();

PublicKey userKeyPub = userKey.getPublic();
```

```
Cipher c = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");

c.init(Cipher.ENCRYPT_MODE, userKeyPub);

byte[] ciphertext  = c.doFinal(plaintext);
```

```
c.init(Cipher.DECRYPT_MODE, privateKey);
byte[] decodificato  = c.doFinal(ciphertext);
```

# Nomi Standard per Primitive

- Descritti nel documento
  - Java Security Standard Algorithm Names

- Sintetizzati nella sezione
  - Security Algorithm Implementation Requirements

DES/CBC/PKCS5Padding
DES/ECB/PKCS5Padding
DESede/ECB/NoPadding
DESede/ECB/PKCS5Padding
RSA/ECB/PKCS1Padding
RSA/ECB/OAEPWithSHA-1AndMGF1Padding

# RSA/ECB ????

- Il modo operativo ECB non ha nessun significato per RSA
  - È utilizzato nei cifrari a blocchi

- Serve per garantire uniformità nell'utilizzo di Cipher.getInstance

# Riferimenti

- Java SE Security
  - http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html
- Java Platform, Standard Edition Security Developer's Guide
  - https://docs.oracle.com/javase/9/security/toc.htm

# Più in dettaglio

- Java Cryptography Architecture Reference Guide
  - https://docs.oracle.com/javase/9/security/java-cryptography-architecture-jca-reference-guide.htm
- Java Security Standard Algorithm Names
  - http://download.java.net/java/jdk9/docs/specs/security/standard-names.html
- Secure Coding Guidelines for Java SE
  - http://www.oracle.com/technetwork/java/seccodeguide-139067.html

# API

- Documentazione Oracle
  - http://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html
  - http://docs.oracle.com/javase/8/docs/api/java/security/package-summary.html