Seguridad en JEE

Sesión1: PKI. Infraestructura de Clave Pública

Índice

- JCA: Java Cryptography Architecture
- JCE: Java Cryptography Extension
- Encriptación Simétrica
- Encriptación Asimétrica
- Autentificación: Firma y Certificado Digital

Índice

- JCA: Java Cryptography Architecture
- JCE: Java Cryptography Extension
- Encriptación Simétrica
- Encriptación Asimétrica
- Autentificación: Firma y Certificado Digital

JCA: Java Cryptography Architecture

- Package `java.security`. Algunas clases:
  - `MessageDigest.` Manejo de digests con SHA-1 o con MD5.
  - `Signature.` Manejo de firma digital con DSA o RSA (encriptación) y MD5 (digests).
  - `Key.` Interfaz para manejo de claves representadas de forma opaca.
  - `KeyFactory.` Convierte representaciones de claves opacas (claves) en transparentes (especificaciones). Es bidireccional.

JCA: Java Cryptography Architecture

- Package `java.security`. Algunas clases más:
  - `CertificateFactory.` Genera certificados y listas de revocación siguiendo una deterinada codificación (p.e. X.509).
  - `KeyPair.` Par (público, privado) para un esquema asimétrico.
  - `KeyPairGenerator.` Obtiene un par (público, privado).
  - `KeyStore.` Interfaz para el manejo de almacenes de claves.
  - `SecureRandom.` Manejo de números aleatorios.

JCA: Java Cryptography Architecture

- Ejemplo. Crear un “digest”

```
MessageDigest md =
  MessageDigest.getInstance(“MD5”,”Sun”);
...
md.update(datos);
byte[] d = md.digest();
```

  - Cada proveedor criptográfico encapsula una serie de implementaciones de mecanismos o algoritmos.
  - La clase `MessageDigest` hace de “proxy” de la clase especificada por el proveedor (Sun en este caso).

Índice

- JCA: Java Cryptography Architecture
- JCE: Java Cryptography Extension
- Encriptación Simétrica
- Encriptación Asimétrica
- Autentificación: Firma y Certificado Digital

JCE: Java Cryptography Extension

- Package `javax.crypto`. Algunas clases:
  - `Cipher.` Cifrador de un determinado algoritmo en modo encriptar/desencriptar.
  - `CipherInput{Output}Stream.` Facilita el uso de cifradores de stream de salida (entrada).
  - `KeyGenerator.` Generador de claves secretas para esquemas simétricos.
  - `SecretKeyFactory.` Convertir claves (opacas) en especificaciones (transparentes) y viceversa.
  - `KeyAgreement.` Funcionalidad para el protocolo del mismo nombre para intercambiar mensajes de forma segura sin intercambiar una clave secreta.
  - `MAC.` Manejo de Message Authentication Codes.

JCE: Java Cryptography Extension

- Ejemplo. Encriptación simétrica
  - Generar la clave “Blowfish” y obtenerla (opaca)

```
KeyGenerator kg =
  KeyGenerator.getInstance("Blowfish");
Key k = KeyGenerator.generateKey();
```

  - Crear e inicializar el cifrador

```
Cipher c = Cipher.
  getInstance("Blowfish/ECB/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE,k);
```

  - Realizar la encriptación (en este caso).

```
byte[] cf = c.doFinal(datos);
```

JCE: Java Cryptography Extension

- Instalación proveedor “Bouncy Castle”
  - Editar $JAVA_HOME/jre/lib/security/java.security

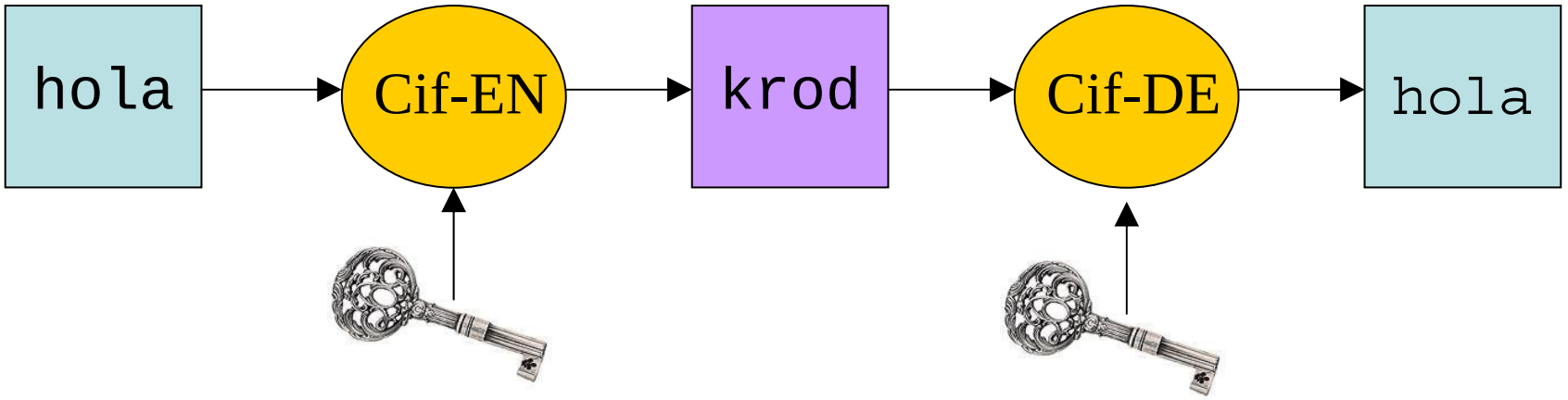
```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.sun.rsajca.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
#Nuevo proveedor de BouncyCastle
security.provider.x=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Índice

- JCA: Java Cryptography Architecture
- JCE: Java Cryptography Extension
- Encriptación Simétrica
- Encriptación Asimétrica
- Autentificación: Firma y Certificado Digital

Estrategias de clave secreta

- Esquema:



Estrategias de clave secreta

- Características:
  - Algoritmos sencillos y rápidos (1000 veces más que los esquemas asimétricos)
  - Aplicar a grandes volúmenes de datos (red, BDs...)
  - Sensible a “ataques-a-mitad” (se expone la clave)
  - Claves de 40-128 bits (típicamente 128)
- Clases de JCE (y métodos):
  - `KeyGenerator: getInstance(),init()`
  - `Key: generateKey()`
  - `Cipher: getInstance(),init(),doFinal()`

“Blowfish” y DESede

- DESede (TripleDES):
  - Evolución de DES que utilizaba claves de 56 bits.
  - Tres rondas (encript-desencript-encript).
  - Se utilizan 2 o 3 claves distintas de 56 bits obteniéndose una clave combinada de 112 o 168 bits.
- “Blowfish”
  - Utiliza claves más seguras (de 448 bits)

“Blowfish” y DESede

- Generar la clave:

```
KeyGenerator generador =
 KeyGenerator.getInstance(“Blowfish");
generador.init (168);
SecretKey clave = generador.generateKey();
```

- Encriptar:

```
Cipher cifrador =
Cipher.getInstance(“Blowfish/ECB/PKCS5Padding”);
cifrador.init(Cipher.ENCRYPT_MODE, clave);
byte[] textoCifrado =
cifrador.doFinal(cadena.getBytes(“UTF8”));
```

“Blowfish” y DESede

- Desencriptar:

```
cifrador.init(Cipher.DECRYPT_MODE, clave);
byte[] textoDescifrado =
cifrador.doFinal(textoCifrado);
String salida =
   new String(textoDescifrado, “UTF8”);
System.out.println(“Salida “ +
textoDescrifado);
```

- Aspectos:
  - Tipo de “Padding”.
  - Modo de encriptación.
  - UTF8 (codificación de bytes).

“Blowfish” y DESede

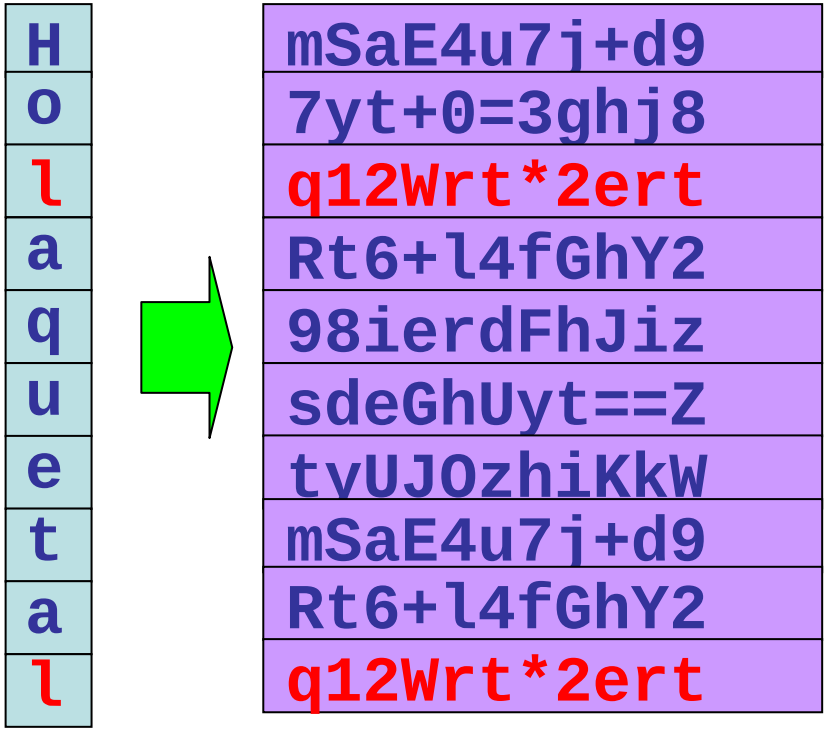
- Tipo de “Padding”
  - Cifradores de bloque operan en trozos de 64 bits.
  - Pero el texto plano no es múltiplo de esta cantidad.
  - Antes de encriptar hay que añadirle complementos para que sea múltiplo.
  - PKCS#5: añadir bytes codificando el número de bytes que se añaden para finalizar el bloque:

H	o	l	a	4	4	4	4
q	u	e	5	5	5	5	5
t	a	l	5	5	5	5	5

Bloques de 8 bytes = 64 bits

“Blowfish” y DESede

- Modo de encriptación:
  - ECB: cifrado uno-a-uno (cifradores de bloque)



  - CBC: lo contrario (cifradores de “stream”)

Blowfish y DESede

```
import javax.crypto.*;

public class Blowfish{

  // Definimos cadena a encriptar
  public static final String cadena = "Esto es un test.";

  public static void main(String[] args) throws Exception {

    System.out.println("El texto original es: " + cadena);

    // Generar una clave Blowfish
    System.out.print("Intentando coger una clave Blowfish....");
    KeyGenerator generador = KeyGenerator.getInstance("Blowfish“);
    generador.init(128);
    SecretKey clave = generador.generateKey();
    System.out.println("OK");

    // Intentar encriptar texto
    System.out.print("Intentando coger un cifrado y encriptar...");
    Cipher cifrador = Cipher.getInstance("Blowfish/ECB/PCKCS5Padding");
    cifrador.init(Cipher.ENCRYPT_MODE, clave);
    byte[] textoCifrado = cifrador.doFinal(cadena.getBytes("UTF8"));
    System.out.println("Ok");
    System.out.println("El texto cifrado es: " + textoCifrado);

    System.out.println("Test completado con exito");
    }
  }
```

Cifradores de “stream”

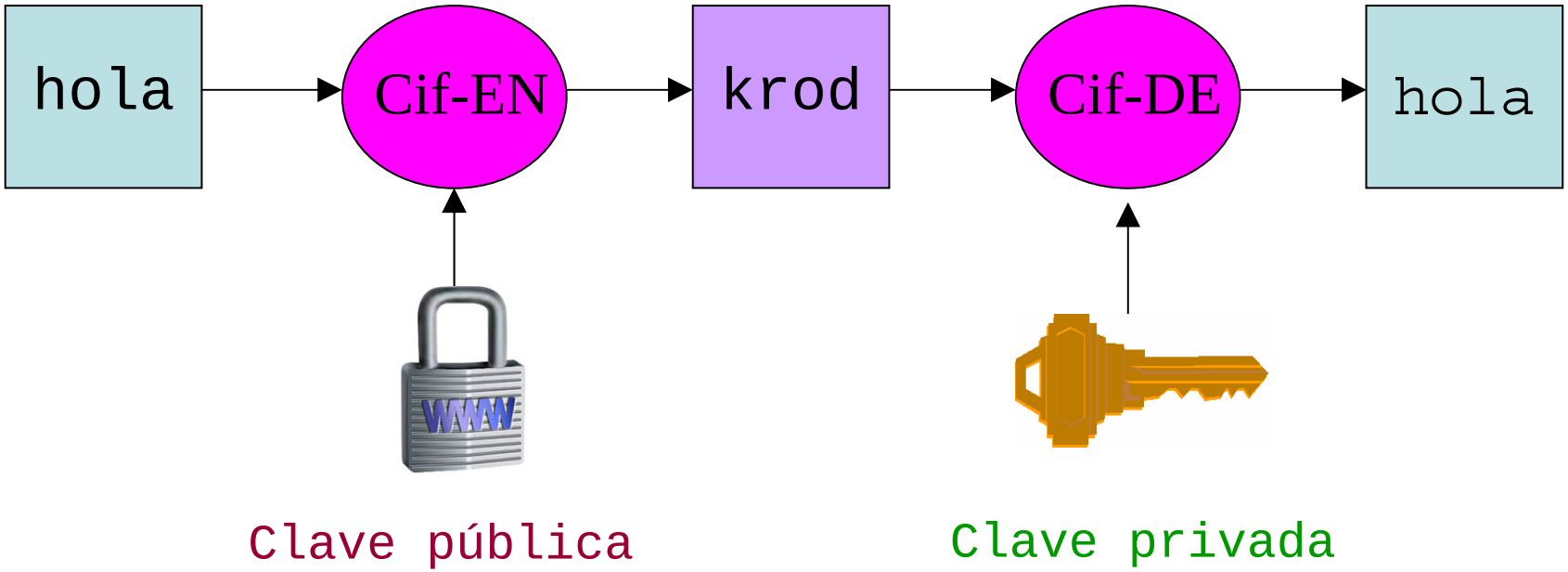
- Motivación:
  - Los cifradores de bloque no son eficientes para encriptar flujos de datos (p.e. “sockets”).
  - En este caso es preferible encriptar “bit a bit” o bien ”byte a byte”
- CBC: Modo de encriptación aconsejado.
  - A cada byte no se le asocia “el mismo código” sino un código que “depende de los bytes anteriores”.
  - La inicialización es aleatoria y se realiza mediante un “Initialization Vector” o IV que debe ser distinto para cada mensaje (aunque no secreto neces.) y debe almacenarse.

Índice

- JCA: Java Cryptography Architecture
- JCE: Java Cryptography Extension
- Encriptación Simétrica
- Encriptación Asimétrica
- Autentificación: Firma y Certificado Digital

Clave pública y clave privada

- Esquema:



Estrategias de clave pública

- Características:
  - Resuelve el problema de compartir un secreto sin que el intercambio sea sensible a “ataques-a-mitad”
  - Las claves (pública y privada) del par son “complementarias” en ambas direcciones
  - Algoritmos: RSA (usado en SSL), El Gammal
  - Claves largas: 1024 bits (equiparable con 128 bits en el caso simétrico.

Clases e interfaces en Java

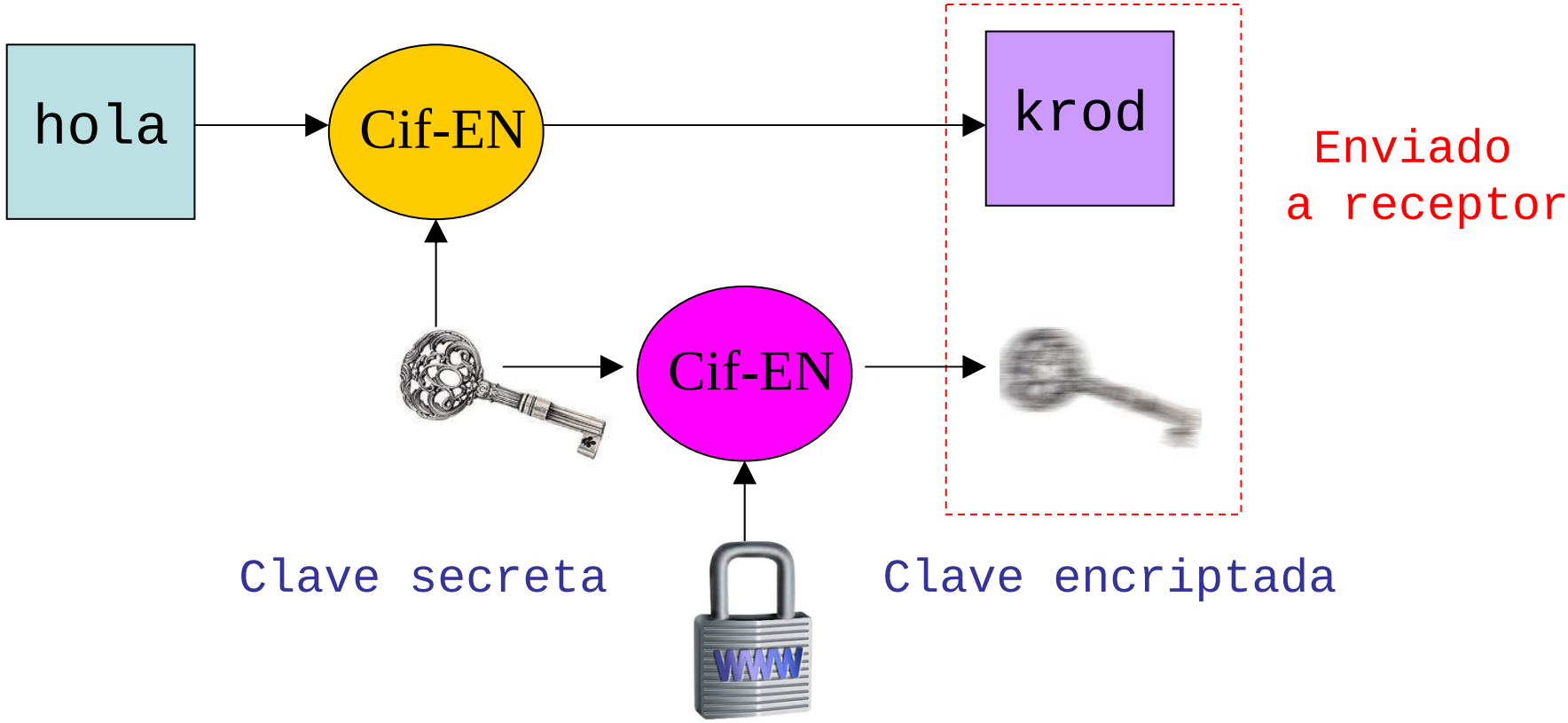
- Clase `KeyPair`
  - Encapsula un par de claves. Para acceder a cada una de ellas se usan los métodos: `getPublic(),getPrivate()`
- Clases `PublicKey(PrivateKey)`
  - Interfaz para claves públicas (privadas).
  - `java.security.interfaces.RSAPublicKey`
  - `RSAPrivateKey`
  - `RSAPrivateCrtKey`
- Clase `KeyPairGenerator`
  - El par se genera con `genKeyPair()`

Encriptación por “session key”

- Motivación:
  - La encriptación asimétrica (que es muy lenta) es especialmente útil para encriptar claves secretas (simétricas).
  - En el modelo “clave-de-sesión” un mensaje se encripta con una clave secreta y esta a su vez se encripta con la clave pública del receptor del mensaje.
  - Cuando el mensaje se recibe, el receptor usa su clave privada para desencriptar la clave secreta y, finalmente, ésta para desencriptar el mensaje.

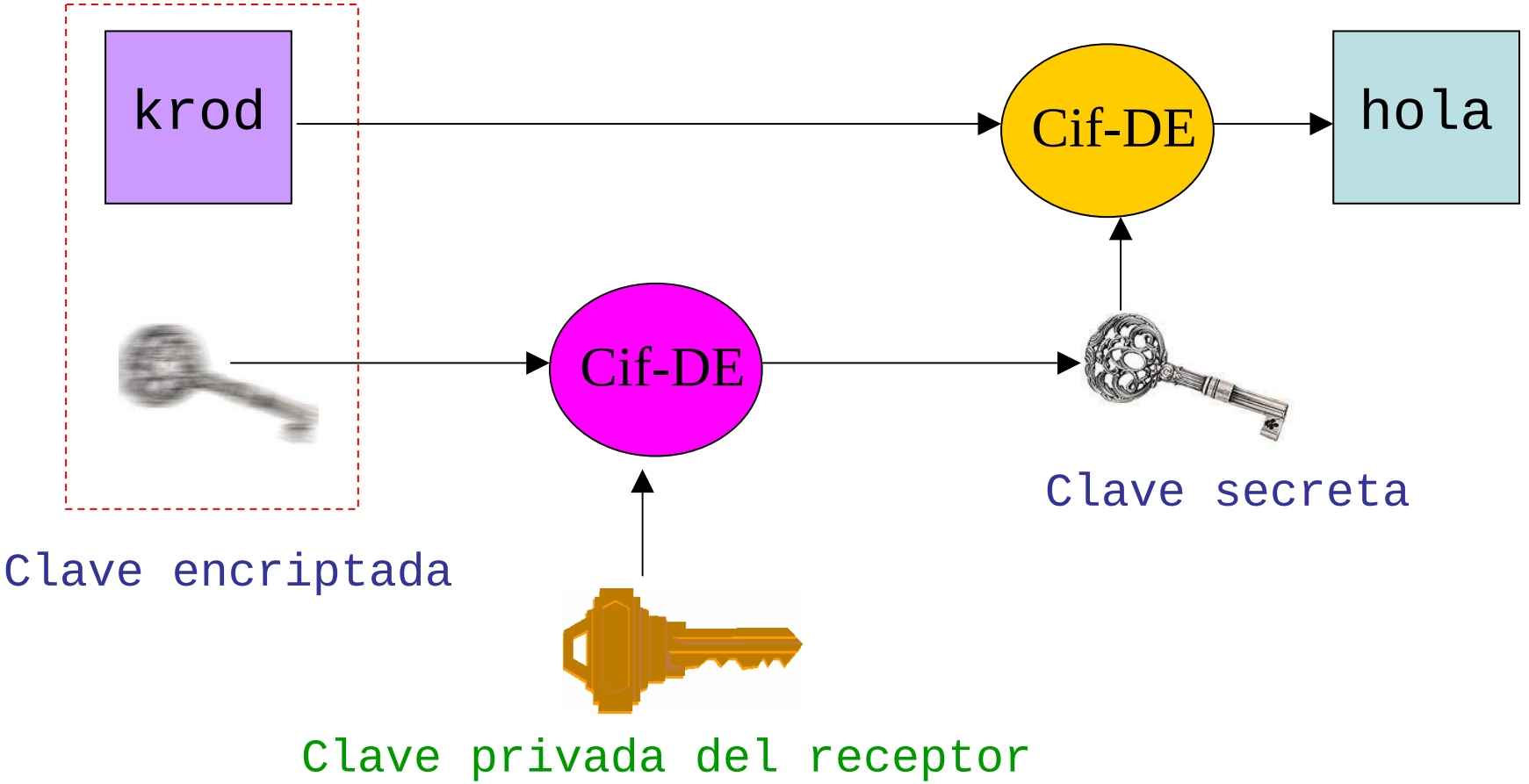
Encriptación por “session key”

- Esquema (encriptación):



Encriptación por “session key”

- Esquema (desencriptación):



Encriptación por “session key”

- En Java…. `RSA.java`
- Clave simétrica (“Blowfish”):

```
System.out.println("Generando clave Blowfish...");
KeyGenerator generador =
 KeyGenerator.getInstance("Blowfish");
generador.init(128);
Key claveBlowfish = generador.generateKey();
System.out.println("Formato:
 "+claveBlowfish.getFormat());
```

Encriptación por “session key”

- Par de claves RSA:

```
System.out.println("Generando par de claves RSA...");
KeyPairGenerator generadorRSA =
  KeyPairGenerator.getInstance("RSA");
generadorRSA.initialize(1024);
KeyPair claves = generadorRSA.genKeyPair();
System.out.println("Generada la clave asimétrica.");
```

Encriptación por “session key”

- Encriptación de la clave “Blowfish”:

```
Cipher cifradorRSA=
  Cipher.getInstance("RSA/ECB/PKCS1Padding");
cifradorRSA.init(Cipher.ENCRYPT_MODE,
  claves.getPublic());

byte[] bytesClaveBlowfish =
  claveBlowfish.getEncoded();
byte[] claveBlowfishCifrada =
  cifradorRSA.doFinal(bytesClaveBlowfish);
```

Encriptación por “session key”

- Desencriptar y recrear de la clave “Blowfish”:

```
cifradorRSA.init(Cipher.DECRYPT_MODE,
  claves.getPrivate());
byte[] bytesClaveBlowfish2 =
  cifradorRSA.doFinal(claveBlowfishCifrada);

SecretKey nuevaClaveBlowfish = new
  SecretKeySpec(bytesClaveBlowfish2, "Blowfish");
```

Índice

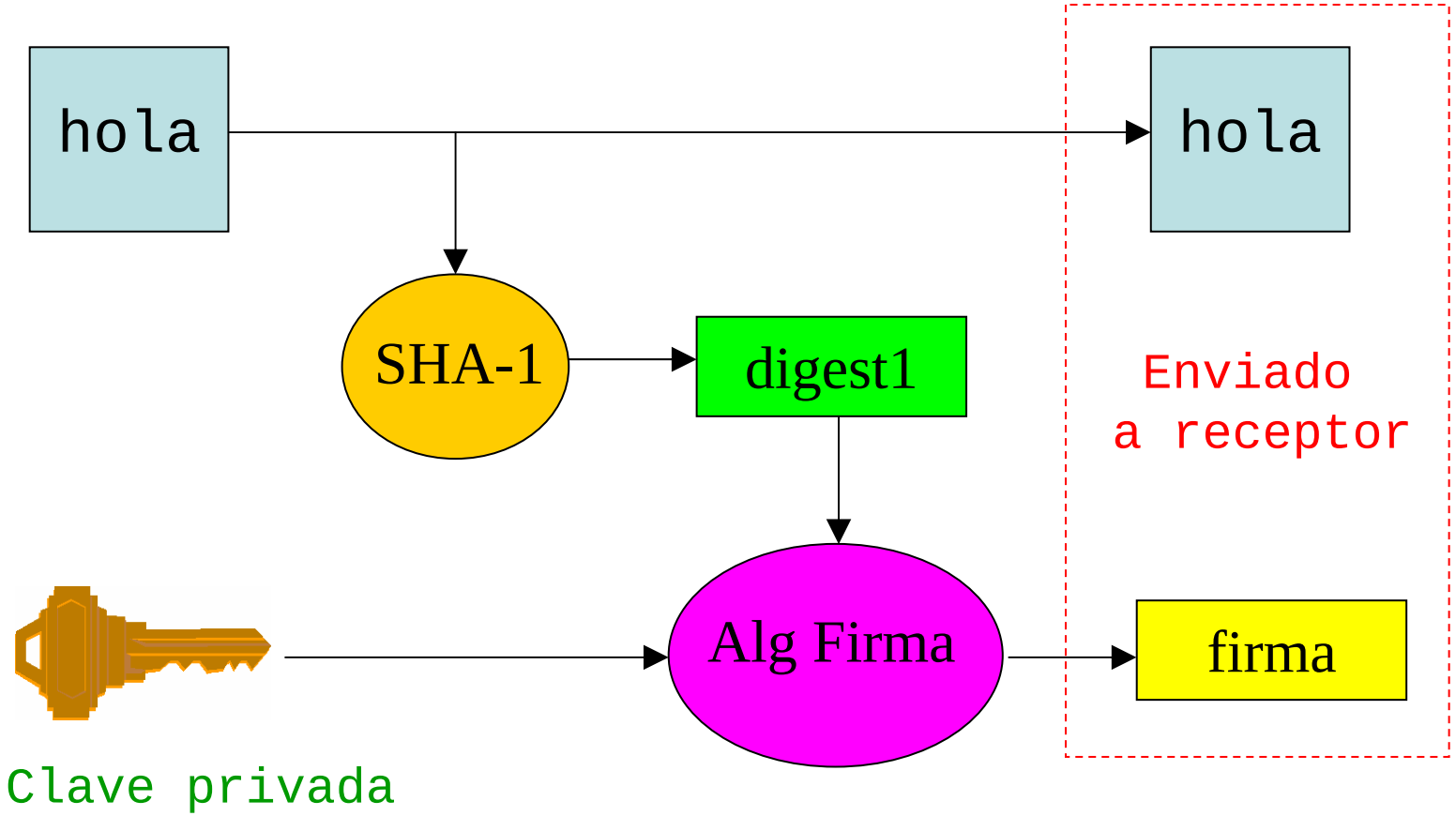
- JCA: Java Cryptography Architecture
- JCE: Java Cryptography Extension
- Encriptación Simétrica
- Encriptación Asimétrica
- Autentificación: Firma y Certificado Digital

Autentificación por firma digital

- Carácterísticas:
  - Firma digital consiste en la asociación de la identidad de un individuo a unos datos (p.e. e-mail)
  - Conceptualmente, una firma no es más que un "digest" procesado por una determinada clave privada asociada al usuario que firma.
  - Pasos a seguir ante la recepción de un mensaje:
    - Obtener el "digest" del mensaje ("digest2")
    - Usar la clave pública del remitente para extraer el digest de la firma ("digest1").
    - Validar sii "digest1" = "digest2"

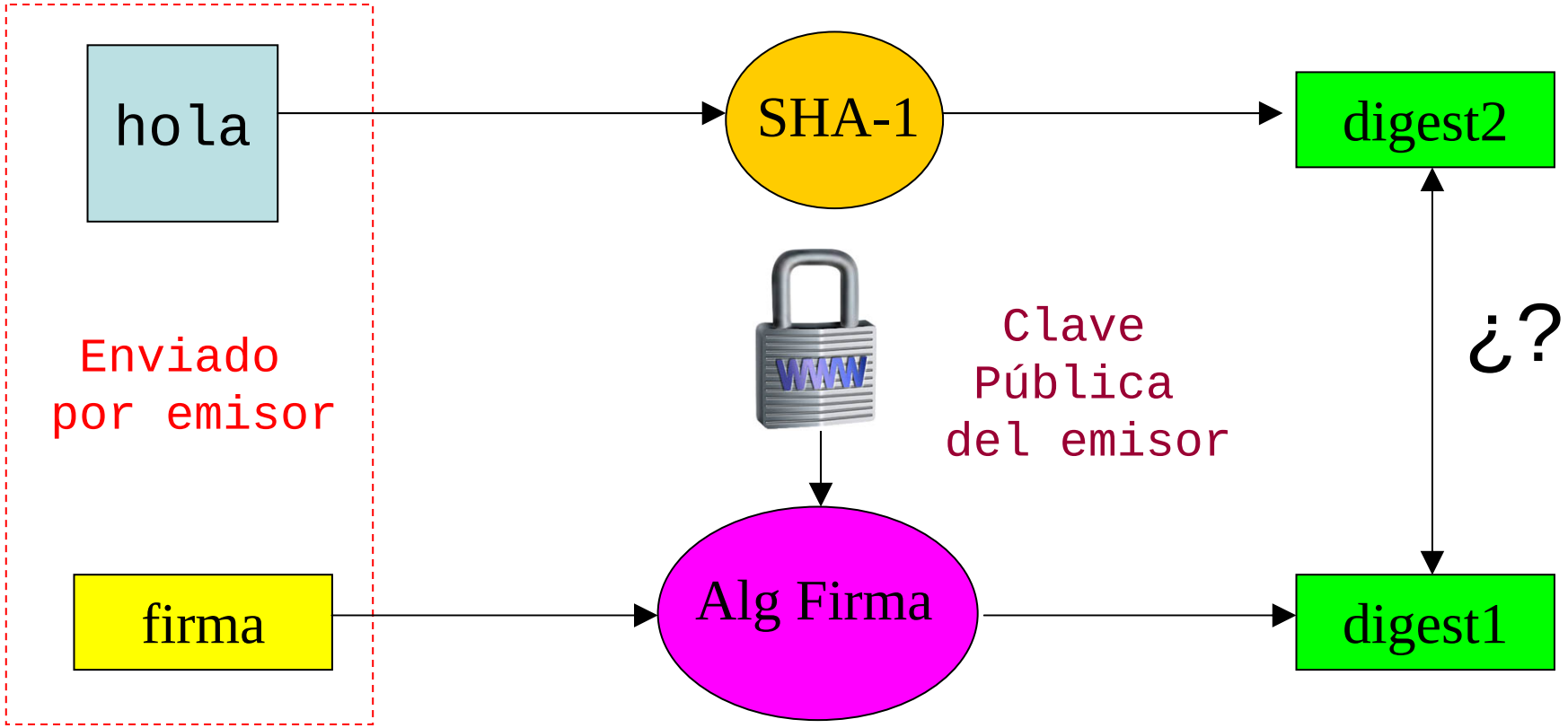
Autentificación por firma digital

- Esquema (emisor):



Autentificación por firma digital

- Esquema (receptor):



Autentificación por firma digital

- Clase `Signature`
  1. Obtener instancia especificando algoritmo: `getInstance()`
  2. Inicializar con una clave privada: `initSign()`
  3. Recogida (preparación) de datos: `update()`
  4. Firmar los datos y devolver firma: `sign()`
  5. Inicializar con clave pública: `initVerify()`
  6. Recogida de datos que fueron firmados: `update()`
  7. Verificación: `verify()`

Autentificación por firma digital

- Ejemplo `FirmaDigital.java`

```
Signature firma =
   Signature.getInstance("MD5WithRSA");
firma.initSign(parClaves.getPrivate());
// Prepara la firma de los datos
firma.update(datos);
// Firmar los datos
byte[] bytesFirma = firma.sign();
```

Autentificación por firma digital

- Ejemplo `FirmaDigital.java`

```
firma.initVerify(parClaves.getPublic());
// Pasar los datos que fueron firmados
firma.update(datos);
// Verificar
boolean verificado = false;
 try {
      verificado = firma.verify(bytesFirma);
    } catch (SignatureException se) {
      verificado = false;
 }
```

Certificados digitales: contenidos

- Motivación:
  - Garantizar que cuando se utiliza una clave pública para validar una firma dicha clave se corresponde con la identidad adecuada.
  - Certificado digital: clave pública + información de identidad firmadas con la clave privada de un tercero (Certificate Authority o CA) Verisign/Thawte
  - Los certificados permiten establecer confianza. Si nuestra MV (JDK) confía (trust) en una determinada CA, entonces aceptará sin reservas todos los certificados que ésta emita.

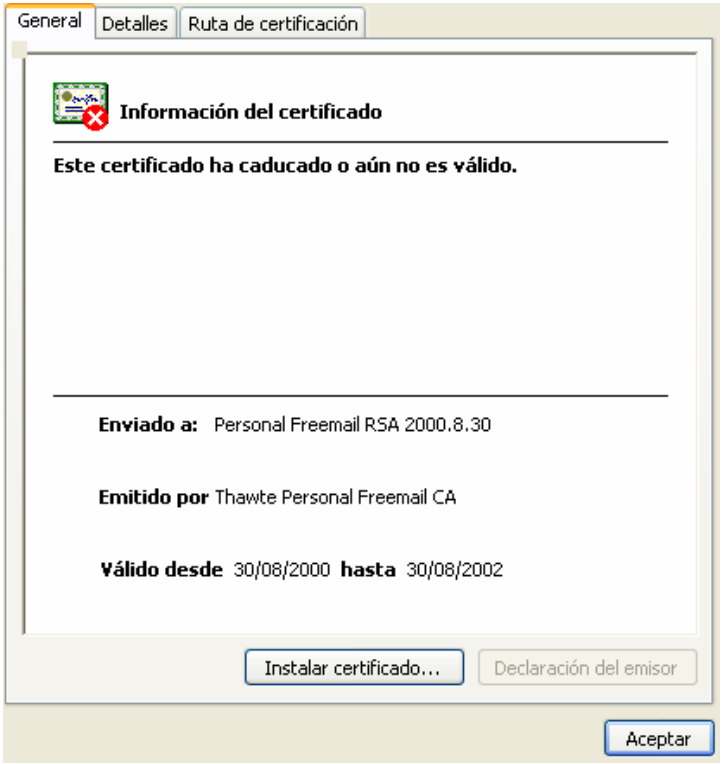
Certificados digitales: contenidos

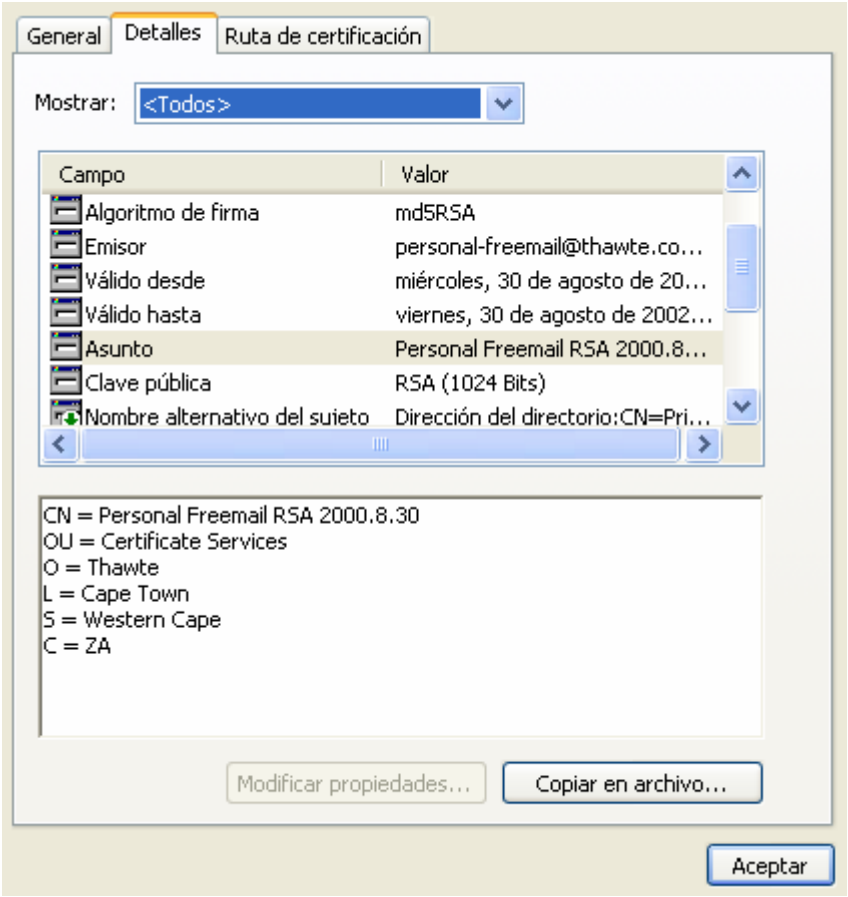
- Contenidos: X.509v1
  - Version: V1, V2, V3
  - Serial Number: Identificador (número entero largo)
  - Signature Algorithm: Algoritmo de firma (p.e. MD5 con RSA)
  - Validity: Intervalo (en fechas) de validez
  - Subject: Identidad X.500 del sujeto en cuestión
  - Subject Public Key: Clave pública del "subject"(asunto)
  - Signature: Firma propiamente dicha (para que podamos compararla frente a la clave pública de la CA).

Certificados digitales: contenidos

- Contenidos: X.509v2
  - Issuer Unique Identifier: Por si el mismo nombre X.500 se usa para diferentes emisores de certificados.
  - Subject Unique Identifier: Por si dos sujetos tienen el mismo nombre X.500
- Contenidos: X.509v3
  - Subject and Issuer Attributes: Info adicional.
  - Key Usage and Policies: Especificar cómo usar el certificado y sus claves (p.e. “solo encriptación”)
  - Certification Path Constraints: Especificar qué certificados pueden crearse a partir de otros.

Certificados digitales: contenidos





Generando certificados: “keytool” y “keystore”

- “keystores” en Java:
  - Almacenes de certificados y claves.
  - “keystore” por defecto: $HOME/.keystore
  - Certificados “trusted” (activos válidos)
  - Claves (simétricas o bien privadas) que pueden usarse para encriptación y/o firma digital. Deben estar asociadas a certificados para esas claves.
  - A pesar de que el acceso al “keystore” está protegido por password es conveniente evitarlos para almacenar claves privadas.

Generando certificados: “keytool” y “keystore”

- keytool: manejo de “keystores”
  - `-certreq:` Petición de certificado para una CA.
  - `-delete:` Borrar entrada del “keystore”
  - `-genkey:` Genera un par de claves para un certificado “auto-firmado”. Se puede especificar el algoritmo con `-keyalg` (p.e. `–keyalg RSA`)
  - `-keystore:` Especifica un fichero como almacén
  - `-printcert:` Mostrar un certificado digital
  - `-selfcert:` Genera un certificado autofirmado
  - `-export:` Exportar certificado de un almacén (en DER).

Generando certificados: “keytool” y “keystore”

- Ejemplos:
  - `keytool –v -list` (Lista las entradas de `.keystore`)
  - `keytool –genkey –alias test` (Añadir una entrada cuyo alias es “test”. La aplicación nos irá pidiendo los datos para completar el certificado)
  - `keytool –export –alias test –file micertificado.cer` (Exporta el certificado cuyo alias es “test” para que sea posteriormente instalable).

Generando certificados: “keytool” y “keystore”

```
Tipo de almacén de claves: jks
Proveedor de almacén de claves: SUN

Su almacén de claves contiene entrada 1

Nombre de alias: test
Fecha de creación: 27-ene-2004
Tipo de entrada: keyEntry
Longitud de la cadena de certificado: 1
Certificado[1]:
Propietario: CN=Francisco Escolano, OU=DCCIA, O=UA, L=Alicante, ST=Alicante, C=ES
Emisor: CN=Francisco Escolano, OU=DCCIA, O=UA, L=Alicante, ST=Alicante, C=ES
Número de serie: 401698e8
Válido desde: Tue Jan 27 17:59:20 CET 2004 hasta: Mon Apr 26 18:59:20 CEST 2004
Huellas digitales del certificado:
             MD5:  BD:1B:89:4C:5D:7B:D3:92:2A:D7:99:21:05:5F:87:AD
             SHA1: 6A:BA:37:07:55:42:E4:A9:18:03:2A:1C:C6:E2:93:BA:30:42:CF:A5


*******************************************
*******************************************
```

Certificados en Java

- Clases de `java.security.cert`
  - `CertificateFactory:` Generador de instancias de objetos `Certificate`
  - `Certificate:` Clase abstracta que encapsula un certificado.
    - `getPublicKey()` devuelve la clave pública del “subject”
    - `verify()` tomando como argumento la clave pública de la CA verifica la firma del certitificado.
  - X509Certificate: Manejo de certificados X.509.
- Imprimir un *.cer: `ImprimirCert.java`

Certificados en Java

```
CertificateFactory factoria =
  CertificateFactory.getInstance("X.509");

   // Abrir el fichero
   FileInputStream fis = new FileInputStream (args[0]);

   // Generar certificado para el fichero
   Certificate cert = factoria.generateCertificate(fis);
   fis.close();

   // Imprimir información
   System.out.println(cert);
```

Certificados en Java

```
[[Version: V1
  Subject: CN=Francisco Escolano, OU=DCCIA, O=UA, L=Alicante, ST=Alicante, C=ES
  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:
    7f85d0b3 a2cc136e e64f6aa0 558ccfaf 1702493c 0abc6cb7 8949bf0f f9394596
    4925a98b bbdc1455 fcba992d 78ba6a58 02e7f216 23f059b7 3575875f af9d281c
    7daf0c78 acbb82ed ea7e8a8c 38ad39cc 92ad579a af470c20 932a6dda 69e985bb
    5303c6d4 59cb3b0b 034d0709 52667938 2ae86398 716450e5 ad6a5fdd ca901dbb

  Validity: [From: Tue Jan 27 17:59:20 CET 2004,
               To: Mon Apr 26 18:59:20 CEST 2004]
  Issuer: CN=Francisco Escolano, OU=DCCIA, O=UA, L=Alicante, ST=Alicante, C=ES
  SerialNumber: [    401698e8]

]
  Algorithm: [SHA1withDSA]
  Signature:
0000: 30 2C 02 14 14 5A 94 5E   68 43 2B 37 35 4D 91 FD  0,...Z.^hC+75M..
0010: 85 BC 34 1D 30 F1 68 7E   02 14 18 21 91 16 BF A7  ..4.0.h....!....
0020: 5C D3 54 6D 21 B2 1A 69   6A 2D 8B 54 6A D4        \.Tm!..ij-.Tj.]
```

Certificados en Java

- Clase `java.security.KeyStore`
  - Manejo de “keystores” por código Java.
  - `load()`: Cargar el “keystore” si le pasamos el nombre de fichero asociado a dicho almacén (p.e. .keystore) y el password asociado.
  - `getCertificate()`: Obtener el objeto `Certificate` asociado a un determinado alias.
- Imprimir desde un determinado “keystore”: `ImprimirCertKS.java`.

Certificados en Java

```
// Abrir el keystore
FileInputStream fIn = new FileInputStream(fich_keystore);
KeyStore keystore = KeyStore.getInstance("JKS");

// Cargar el keystore
keystore.load(fIn, password);

// Obtener el certificado
Certificate cert = keystore.getCertificate(alias);

// Mostrar el certificado
System.out.println(cert);
```

Crear nuestra propia CA

- Motivación:
  - Parte de la infraestructura de clave pública (PKI).
  - Queremos emitir nuestros propios certificados:
    - E-mail: emitir certificados de e-mail para que todos los miembros de nuestra organización puedan firmar su correo.
    - Crear certificados para validar el acceso de nuestros clientes a nuestras bases de datos.
  - Usar el paquete `sun.security.x509` (aunque no se recomienda en general puesto que no es 100% Java y porque pueden causar interferencias de seguridad).

Crear nuestra propia CA

- Clases `sun.security.x509`
  - `X509CertImpl:` Implementación de un `Certificate` X.509 que podremos firmar con `sign()`
  - `X509CertInfo:` Encapsula los atributos del certificado y permite instanciarlos en el momento de crearlo.
  - `X500Name:` Nombre X.500 (CN,OU,….)
  - `AlgorithmId:` Identifica algoritmo criptográfico
  - `CertificateSubjectName, CertificateValidity, CertificateSerialNumber, CertificateIssuerName, CertificateAlgorithmID,`

Crear nuestra propia CA

Pasos a seguir:

1. Crear certificado y clave privada para nuestra CA:

```
keytool –genkey –v -alias CA –keyalg RSA –keystore almacen
```

2. Crear un certificado que queramos firmar

```
keytool -genkey –v alias miClave –keyalg RSA –keystore almacen
```

3. Reemplazar el certificado que acabamos de crear (que es auto-firmado) por otro que sea firmado por nuestra CA, usando el certificado y la clave privada de la CA. Ver el ejemplo: `FirmarCertificado.java`

Crear nuestra propia CA

- `FirmarCertificado.java`
  1. Leer clave privada y certificado de la CA:

```
PrivateKey clavePrivadaCA =
(PrivateKey)keystore.getKey(aliasCA,
passwordCA);

java.security.cert.Certificate certificadoCA =
keystore.getCertificate(aliasCA);
```

Crear nuestra propia CA

- FirmarCertificado.java

2. Crear una implementación X.509 para el certificado de la CA:

```
byte[] codificado = certificadoCA.getEncoded();
X509CertImpl implementacionCA = new
X509CertImpl(codificado);

X509CertInfo infoCA =
(X509CertInfo)implementacionCA.get
(X509CertImpl.NAME + "." + X509CertImpl.INFO);
X500Name emisorCA = (X500Name)infoCA.get
(X509CertInfo.SUBJECT + "." +
CertificateIssuerName.DN_NAME);
```

Crear nuestra propia CA

- `FirmarCertificado.java`

3. Leer la clave privada y el certificado a firmar:

```
java.security.cert.Certificate cert =
keystore.getCertificate(aliasCert);

PrivateKey clavePrivada =
(PrivateKey)keystore.getKey(aliasCert,
passwordCert);
```

Crear nuestra propia CA

- `FirmarCertificado.java`

4. Crear de nuevo una implementación X.509 para el certificado a firmar:

```
codificado = cert.getEncoded();

X509CertImpl implementacionCert = new
X509CertImpl(codificado);

X509CertInfo infoCert =
(X509CertInfo)implementacionCert.get

   (X509CertImpl.NAME + "." +
X509CertImpl.INFO);
```

Crear nuestra propia CA

5. Especificar y almacenar el período de validez:

```
Date inicio = new Date();

Date fin = new Date(inicio.getTime() +
VALIDEZ*24*60*60*1000L);

CertificateValidity intervalo = new
CertificateValidity(inicio, fin);
```

6. Crear y almacenar el número de serie:

```
infoCert.set(X509CertInfo.SERIAL_NUMBER, new
CertificateSerialNumber((int)(inicio.getTime()/
1000)));
```

Crear nuestra propia CA

5. Especificar y almacenar el período de validez:

```
Date inicio = new Date();

Date fin = new Date(inicio.getTime() +
VALIDEZ*24*60*60*1000L);

CertificateValidity intervalo = new
CertificateValidity(inicio, fin);
```

6. Crear y almacenar el número de serie:

```
infoCert.set(X509CertInfo.SERIAL_NUMBER, new
CertificateSerialNumber((int)(inicio.getTime()/
1000)));
```

Crear nuestra propia CA

9,10. Crear el nuevo certificado y firmarlo:

```
X509CertImpl nuevoCert = new X509CertImpl(infoCert);

nuevoCert.sign(clavePrivadaCA, ALG);
```

11,12. Almacenar en el “keystore” y éste en un fichero:

```
keystore.setKeyEntry(aliasNuevo, clavePrivada,
passwordCert, new java.security.cert.Certificate[]
{ nuevoCert } );

FileOutputStream output = new
FileOutputStream(fich_keystore);
keystore.store(output, password); output.close();
```

Crear nuestra propia CA

Para aplicar esta firma:

```
java FirmarCertificado almacen CA miClave miClave2
```

Para ver el nuevo certificado:

```
keytool –list –v –keystore almacen
```

Para exportar certificado de la CA a .crt:

```
keytool –export –alias CA –keystore almacen –
   file CA.crt
```

Crear nuestra propia CA

```
Nombre de alias: miclave2
Fecha de creación: 28-ene-2004
Tipo de entrada: keyEntry
Longitud de la cadena de certificado: 1
Certificado[1]:
Propietario: CN="Francisco ", OU=j2ee, O=Unknown, L=Unknown, ST=Unknown,
C=Unknown
Emisor: CN=CA, OU=j2ee, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Número de serie: 4017e829
Válido desde: Wed Jan 28 17:49:45 CET 2004 hasta: Thu Jan 27 17:49:45 CET
2005
Huellas digitales del certificado:
         MD5:  2A:47:49:9B:A5:22:06:4C:7A:63:FD:64:4D:75:42:BD
         SHA1: CD:00:93:64:3B:61:B0:A8:2A:3F:D1:32:DC:2D:B0:7B:AB:EB:01:44


*******************************************
*******************************************
```

Ejercicios…

- Encriptación de tarjetas de crédito:
  - Dado el proyecto adjunto incluid las modificaciones oportunas para lograr la siguiente funcionalidad.

1.
2. Un cliente publica los mensajes 1,2 y 3 en ese topico
3. El suscriptor los lee y después se cierra.
4. El cliente publica los mensajes 4, 5 y 6 en ese tópico.
5. El subscriptor de crea de nuevo. Leerá los mensajes correspondientes.
6. El subsciptor se cierra.
7. La subscripción duradera se cancela.

Encriptación de tarjetas de crédito

- Supuesto:
  - Tenemos un servidor que acepta datos de tarjetas y los almacena en una BD encriptada con una clave pública.
- Clases y objetos java:
  - `CreditCard()`: Objeto para codificar tarjetas
    - `mAccountID`: con `getAccountID()`
    - `mCreditCardNumber`: con `getCreditCardNumber()`
  - `CreditCardDBO()`: Tarjetas encriptadas y "session-key"
    - `mAccountID: getAccountID()`
    - `mEncryptedCCNumber: getEncryptedCCNumber()`
    - `mEncryptedSessionKey: getEncryptedSessionKey()`

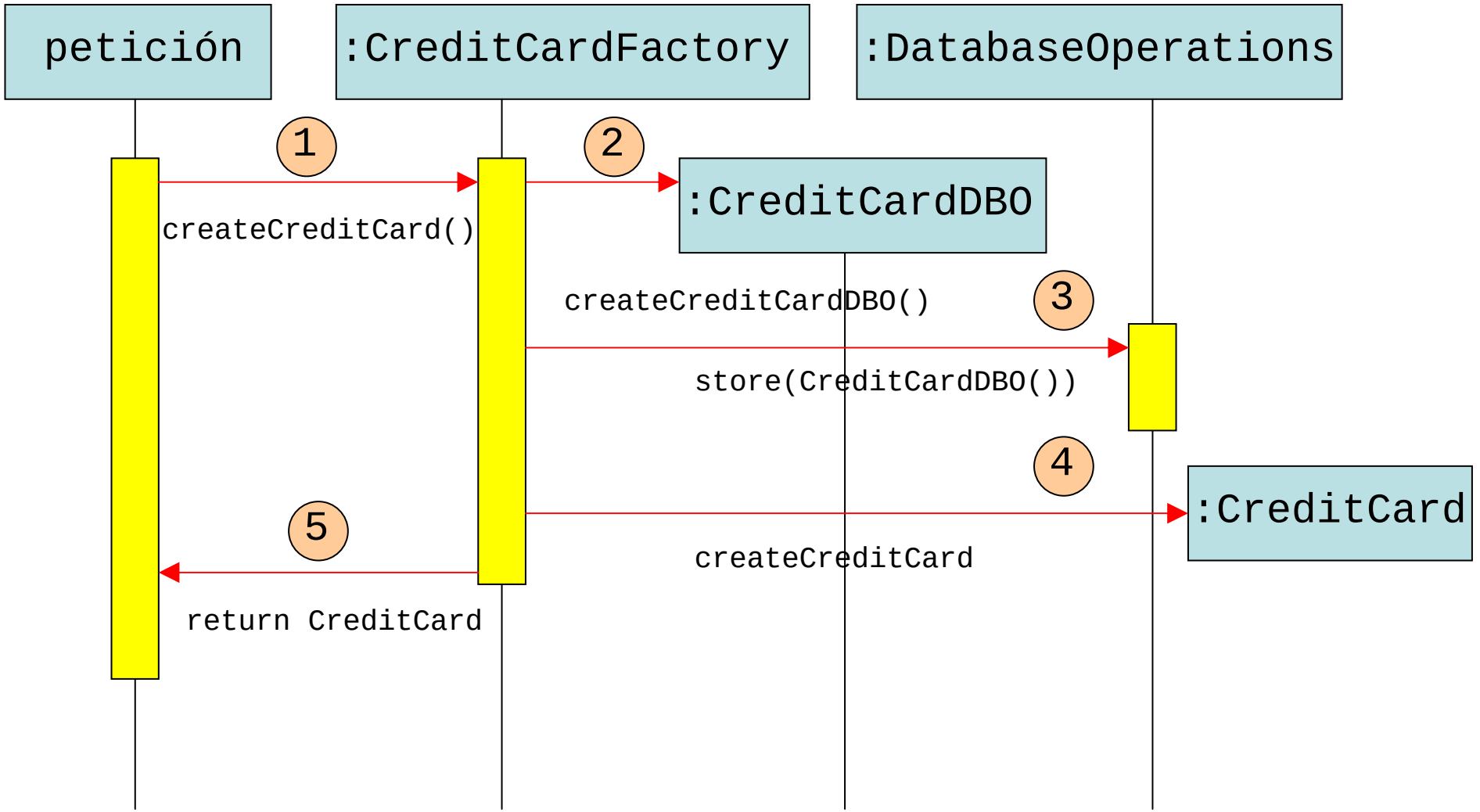
Encriptación de tarjetas de crédito

- Clases y objetos java:
  - `DatabaseOperation:` Maneja acceso a JDBC mediante las siguientes operaciones definidas y sin encriptación:
    - `getAllCreditCardAccountIDs()`
    - `loadCreditCardDBO()`
    - `getCreditCardNumber()`
  - `CreditCardFactory:` Maneja la encriptación y desecriptación de tarjetas utilizando una `mPublicKey` para encriptar. Dada la clave privada permite:
    - `createCreditCard()`
    - `findAllCreditCards()`
    - `findCreditCard()`

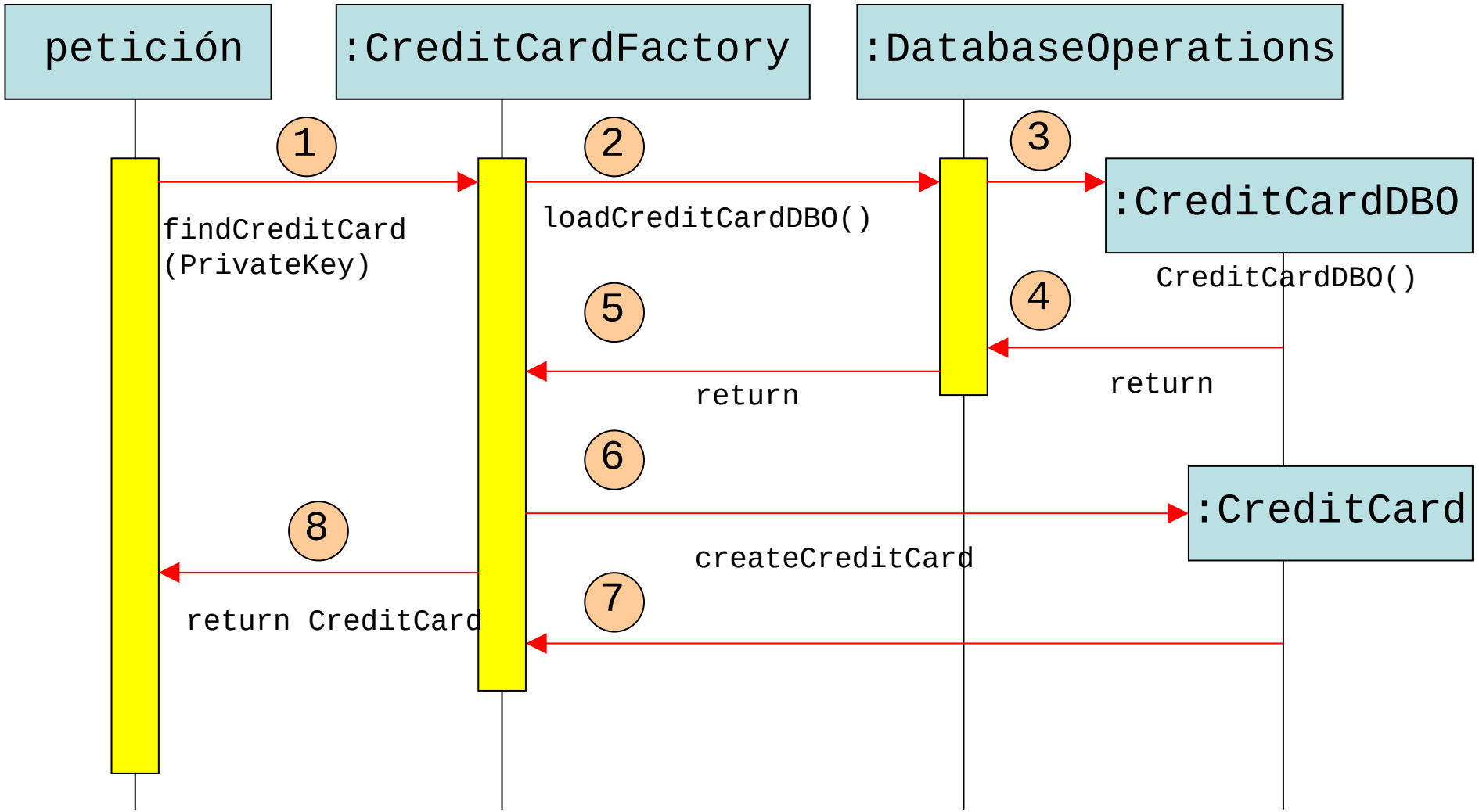
Configuración de la BD

```
CREATE DATABASE projava;
USE projava;
CREATE TABLE account (
   account_id INT8 PRIMARY KEY,
   customer_name VARCHAR(40),
   balance FLOAT,
   cert_serial_number VARCHAR(255)
);
CREATE TABLE credit_card (
  account_id INT8 PRIMARY KEY,
  session_key VARCHAR(255),
  cc_number VARCHAR(100) );
```

createCreditCard



findCreditCard



Modificando y consulando la BD

- Aplicación: `CreateTest.java`
  - cd a directorio con los ficheros:
    - config.properties, publica.cer tarjetas.ks
    - Ficheros *.java
    - Fichero *.jar del JDBC instalado o accesible
  - Compilar y probar:
    - $javac –d . *.java y sale com.projavasecurity.ecommerce
    - $ java com.projavasecurity.ecommerce.CreateTest

      1 “1234 5678 9012 3456”
  - Consultar desde mysql:
    - USE projava;
    - SELECT * FROM credit_card;

Modificando y consultando la BD

- Aplicación: `ViewTest.java`
  1. Consultar el keystore tarjetas.ks (saber password) y consultar la “clave privada” (para desencriptar)
  2. CreditCardFactory y `findAllCreditCards`.
  - Compilar y probar:
    - $javac –d . *.java y sale com.projavasecurity.ecommerce
    - $ java com.projavasecurity.ecommerce.ViewTest
    - Obtendremos la tarjeta desencriptada.