Proveedores de servicios criptográficos

El API **JCA** (*Java Cryptography Architecture*, incluye la extensión criptográfica de **Java JCE** – *Java Cryptography Extension*) incluída dentro del paquete JDK incluye dos componentes de software:

- El marco que define y apoya los servicios criptográficos para que los proveedores faciliten implementaciones. Este marco incluye paquetes como
    - java.security
    - javax.crypto
    - javax.crypto.spec
    - javax.crypto.interfaces
- Los proveedores reales, tales como Sun, SunRsaSign, SunJCE, que contienen las implementaciones criptográficas reales. El proveedor es el encargado de proporcionar la implementación de uno o varios algoritmos al programador. Los proveedores de seguridad se definen en el fichero **java.security** localizo en la carpeta **java.home\lib\security**. Forman una lista de entradas con un número que indican el orden de búsqueda cuando en los programas no se especifica un proveedor.
    - security.provider.1=sun.security.provider.Sun
    - security.provider.2=sun.security.rsa.SunRsaSign
    - security.provider.3=com.sun.net.ssl.internal.ssl.Provider
    - security.provider.4=com.sun.crypyo.provider.SunJCE
    - security.provider.5=sun.security.jgss.SunProvider

JCA define el concepto de proveedor mediante la clase **Provider** del paquete **java.security**. Se trata de una clase abstracta que debe ser redefinida por clases proveedor específicas.

Tiene métodos para acceder a informaciones sobre las implementaciones de los algoritmos para la generación, conversión y gestión de claves y la generación de firmas y resúmenes, como el nombre del proveedor, el número de versión, etc.

Resúmenes de mensajes

Un *message digest* o resúmen de mensajes (también se le conoce como **función hash**) es una marca digital de un bloque de datos.

La clase **MessageDigest** permite a las aplicaciones implementar algoritmos de resumen de mensajes, como **MD5**, **SHA-1** o **SHA-256**. Dispone de un constructor protegido, por lo que se accede a él mediante el método ***getInstance(String algoritmo)***.

Algunos métodos de la clase **MessageDigest** son:

MÉTODOS	MISIÓN
public static MessageDigest getInstance(String algoritmo) public static MessageDigest getInstance(String algoritmo, String proveedor)	Devuelve un objeto *MessageDigest* que implementa el algoritmo de resumen especificado En el primer caso, los proveedores de seguridad se buscan según el orden establecido en el fichero **java.security**. En el segundo caso se busca el proveedor dado. Nombres válidos para el proveedor de seguridad predeterminado de *Sun* son SHA, SHA-1 y MD5 Puede lanzar la excepción *NoSuchAlgorithmException* si no hay proveedor que implemente el algoritmo dado. Si el nombre de proveedor no se encuentra se produce *NoSuchProviderException*
void update(byte input)	Realiza el resumen del byte especificado
void update(byte[] input)	Realiza el resumen del array de bytes especificado
byte[] digest()	Completa el cálculo del valor hash, devuelve el resumen obtenido
byte [] digest (byte [] entrada)	Realiza una actualización final sobre el resumen utilizando el array de bytes indicado en el argumento, y luego completa el cálculo de resumen
void reset()	Reinicializa el objeto resumen para un nuevo uso
int getDigestLength()	Devuelve la longitud del resumen en bytes, o 0 si la operación no está soportada por el proveedor
String getAlgorithm()	Devuelve un String que identifica el algoritmo
Provider getProvider()	Devuelve el proveedor del objeto
static boolean isEqual(byte[] digesta, byte[] digestb)	Comprueba si dos mensajes resumen son iguales. Devuelve true si son iguales y false en caso contrario

Ejemplo (en el adjunto de ejercicios=

Archivo EJEMPLO4.JAVA

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.Provider;

public class Ejemplo4 {

	public static void main(String[] args) {

		MessageDigest md;
		try {
			md = MessageDigest.getInstance("SHA");
			String texto = "Esto es un texto plano.";

			byte dataBytes[] = texto.getBytes();//TEXTO A BYTES
			md.update(dataBytes) ;//SE INTRQDUCE TEXTO EN BYTES
A RESUMIR
			byte resumen[] = md.digest();//SE CALCULA EL RESUMEN

			//PARA CREAR UN RESUMEN CIFRADO CON CLAVE
			//String clave="clave de cifrado";
			//byte resumen[] = md.digest(clave.getBytes()); //
SE CALCULA EL RESUMEN CIFRADO


			System.out.println("Mensaje original: " + texto);
			System.out.println("Número de bytes: " +
md.getDigestLength());
			System.out.println("Algoritmo: " +
md.getAlgorithm());
			System.out.println("Mensaje resumen: " + new
String(resumen));
			System.out.println("Mensaje en Hexadecimal: " +
Hexadecimal(resumen));
			Provider proveedor = md.getProvider();
			System.out.println("Proveedor: " +
proveedor.toString());
		} catch (NoSuchAlgorithmException e) {
e.printStackTrace(); }

	}//Fin de main


	// CONVIERTE UN ARRAY DE BYTES A HEXADECIMAL
	static String Hexadecimal(byte[] resumen) {

		String hex = "";
		for (int i = 0; i < resumen.length; i++) {

			String h = Integer.toHexString(resumen[i] & 0xFF);
			if (h.length() == 1)
				hex += "0";
			hex += h;
		}//Fin de for

		return hex.toUpperCase();
```

```
      }// Fin de Hexadecimal

}//Fin de Ejemplo4
```

Genera el resumen de un texto plano. Con el método `MessageDigest.getInstance("SHA")` se obtiene una instancia del algoritmo SHA. El texto plano lo pasamos a un array de bytes y el array se pasa como argumento al método `upate()`, finalmente con el método `digest()` se obtiene el resumen del mensaje. Después se muestra en pantalla el número de bytes generados en el mensaje, el algoritmo utilizado, el resumen generado y convertido a Hexadecimal y por último información del proveedor.

Se puede crear un resumen cifrado con clave usando el segundo método `digest(bytes[])`, donde se proporciona la clave en un array de bytes.

```
String clave="clave de cifrado";
byte dataBytes[] = texto.getBytes(); //TEXTO A BYTES
md.update(dataBytes) ; // SE INTRODUCE TEXTO EN BYTES A RESUMIR
byte resumen[] = md.digest(clave.getBytes());  // SE CALCULA EL
RESUMEN CIFRADO
```

Segundo ejemplo

Archivo EJEMPLO5.JAVA

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class Ejemplo5 {

      public static void main(String args[]) {

            try {
                  FileOutputStream fileout = new
FileOutputStream("DATOS.DAT");
                  ObjectOutputStream dataOS = new
ObjectOutputStream(fileout);
                  MessageDigest md = MessageDigest.getInstance("SHA");
                  String datos = "En un lugar de la Mancha, "
                              + "de cuyo nombre no quiero acordarme, no
ha mucho tiempo "
                              + "que vivía un hidalgo de los de lanza
en astillero, "
                              + "adarga antigua, rocín flaco y galgo
corredor.";
                  byte dataBytes[] = datos.getBytes();
                  md.update(dataBytes) ;// TEXTQ A RESUMIR
                  byte resumen[] = md.digest(); // SE CALCULA EL
RESUMEN
                  dataOS.writeObject(datos); //se escriben los datos
                  dataOS.writeObject(resumen);//Se escribe el resumen
                  dataOS.close();
                  fileout.close();
            } catch (IOException e) { e.printStackTrace();
            } catch (NoSuchAlgorithmException e) {
e.printStackTrace(); }

      }//Fin de main
}//Fin de Ejemplo5
```

Guardaremos un mensaje en un fichero.
También guardaremos en el fichero el resumen del mensaje, para asegurarnos de que a la hora de leer el mensaje el fichero **no esté dañado o no haya sido manipulado y los datos sean los correctos**.

Tercer ejemplo

Archivo EJEMPLO6.JAVA

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;

import java.security.MessageDigest;

public class Ejemplo6 {

	public static void main(String args[]) {

		try {
			FileInputStream fileout = new
FileInputStream("DATOS.DAT");
			ObjectInputStream dataOS = new
ObjectInputStream(fileout);
			Object o = dataOS.readObject();

			// Primera lectura, se obtiene el String
			String datos = (String) o;
			System.out.println("Datos: " + datos);

			// Segunda lectura, se obtiene el resumen
			o = dataOS.readObject();
			byte resumenOriginal[] = (byte[]) o;

			MessageDigest md = MessageDigest.getInstance("SHA");
			//Se calcula el resumen del String leído del fichero
			md.update(datos.getBytes());// TEXTO A RESUMIR
			byte resumenActual[] = md.digest(); // SE CALCULA EL
RESUMEN

			//Se comprueban lo dos resúmenes
			if (MessageDigest.isEqual(resumenActual,
resumenOriginal))
				System. out.println ("DATOS VÁLIDOS") ;
			else
				System.out.println("DATOS NO VÁLIDOS") ;
			dataOS.close();
			fileout.close();
		}catch (Exception e) { e.printStackTrace(); }

	}//Fin de main
}//Fin de Ejemplo6
```

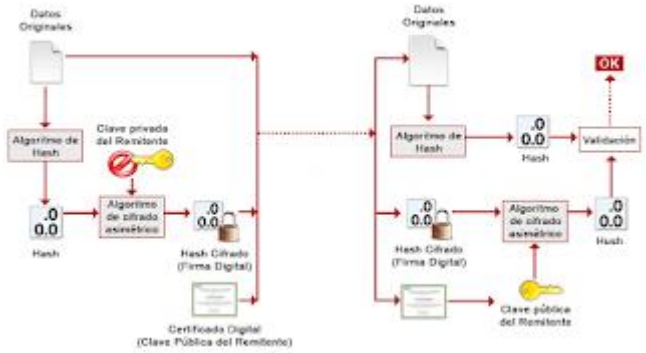
Al recuperar los datos del fichero primero necesitamos leer el String y luego el resumen, a continuación hemos de calcular de nuevo el resumen con el String leído y comparar este resumen con el leído del fichero.

Generando y verificando firmas digitales

El resumen de un mensaje no nos da un alto nivel de seguridad.

Se puede decir que el fichero no es correcto si el texto que se lee no produce la misma salida que el resumen guardado.

Pero alguien puede cambiar el texto y el resumen, y no podemos estar seguros de que el texto sea el que debería ser.



Clase KeyPairGenerator

En algunos casos, el par de claves (**clave pública** y **clave privada**) están disponibles en ficheros. En ese caso, el programa puede importar y utilizar la clave privada para firmar.

En otros casos, el programa necesita generar el par de claves.

La clase KeyParGenerator nos permite generar el par de claves. Dispone de un constructor protegido, por lo que se accede a él mediante el método **`getInstance(String algoritmo)`**.

MÉTODOS	MISIÓN
static KeyPairGenerator getInstance(String algoritmo) static KeyPairGenerator getInstance(String algoritmo, String provider)	Devuelve un objeto *KeyPairGenerator* que genera un par de claves pública/privada para el algoritmo especificado. Puede lanzar la excepción *NoSuchAlgorithmException* En el segundo método se especifica el proveedor. Si el nombre de proveedor no se encuentra se produce *NoSuchProviderException*
void initialize(int keysize, SecureRandom random)	Inicializa el generador de par de claves para un determinado tamaño de clave y un generdor de números aleatorios
KeyPair generateKeyPair() KeyPair genKeyPair()	Genera el par de claves

Clase KeyPair

Es una clase soporte para generar las claves pública y privada.
Dispone de dos métodos:

MÉTODOS	MISIÓN
PrivateKey getPrivate()	Devuelve una referencia a la clave privada del par de claves
PublicKey getPublic()	Devuelve una referencia a la clave pública del par de claves

PrivateKey y **PublicKey** son interfaces que agrupan todas las interfaces de clave privada y pública respectivamente.

Clase Signature

Se usa para firmar los datos.
Un objeto de esta clase se puede utilizar para generar y verificar firmas digitales.

Dispone de un constructor protegido y se acceder a él mediante el método **`getInstance(String algoritmo)`**.

Algunos de sus métodos:

MÉTODOS	MISIÓN
static Signature getInstance(String algoritmo)	Devuelve un objeto *Signature* que implementa el algoritmo especificado. Puede lanzar la excepción *NoSuchAlgorithmException*
static Signature getInstance(String algoritmo, String provider)	En el segundo método se especifica el proveedor. Si el nombre de proveedor no se encuentra se produce *NoSuchProviderException*
void initSign(PrivateKey privateKey, SecureRandom random)	Inicializa el objeto para la firma. Se especifica la clave privada de la identidad cuya firma se va a generar y la fuente de aleatoriedad. Si la clave no es válida puede lanzar la excepción *InvalidKeyException*
void update(byte b)	Actualiza los datos a firmar o verificar usando el byte especificado
void update(byte[] data)	Actualiza los datos a firmar o verificar usando el array de bytes especificado
void update(ByteBuffer data)	Actualiza los datos a firmar o verificar usando el ByteBuffer especificado
byte[] sign()	Devuelve en un array de bytes la firma de los datos
void initVerify(PublicKey publicKey)	Inicializa el objeto para la verificación de la firma. Necesita como parámetro la clave pública. Si la clave no es válida puede lanzar la excepción *InvalidKeyException*
boolean verify(byte[] signature)	Verifica la firma que se pasa como parámetro

Al especificar el nombre del algoritmo de firma, también se debe incluir el nombre del algoritmo de resumen de mensajes utilizado por el algoritmo de firma. **SHA1withDSA** es una forma de especificar el algoritmo de firma DSA, usando el algoritmo de resumen SHA-1. **MD5withRSA** significa algoritmo de resumen MD5 con algoritmo de firma RSA.

Existen tres fases en el uso de un objeto **Signature** ya sea para firmar o verificar los datos: inicialización (ya sea con clave pública **initVerify()** o clave privada **initSign()**), actualización (**update()**) y firma (**sign()**) o verificación (**verify()**).

Ejemplo

Archivo EJEMPLO7.JAVA

```
import java.security.*;

public class Ejemplo7 {

	public static void main(String[] args) {

		try {
			KeyPairGenerator keyGen =
KeyPairGenerator.getInstance("DSA");
			//SE INICIALIZA EL GENERADOR DE CLAVES
			SecureRandom numero =
SecureRandom.getInstance("SHA1PRNG");
			keyGen.initialize (1024, numero);

			//SE CREA EL PAR DE CLAVES PRIVADA Y PÚBLICA
			KeyPair par = keyGen.generateKeyPair();
			PrivateKey clavepriv = par.getPrivate();
			PublicKey clavepub = par.getPublic();

			//FIRMA CON CLAVE PRIVADA EL MENSAJE
			//AL OBJETO Signature SE LE SUMINISTRAN LOS DATOS A
FIRMAR
			Signature dsa =
Signature.getInstance("SHAlwithDSA");
			dsa.initSign (clavepriv);
			String mensaje = "Este mensaje va a ser firmado";
			dsa.update(mensaje.getBytes());

			byte [] firma= dsa.sign(); //MENSAJE FIRMADO

			//EL RECEPTOR DEL MENSAJE
			//VERIFICA CON LA CLAVE PÚBLICA EL MENSAJE FIRMADO
			//AL OBJETO signature SE LE SUMINIST. LOS DATOS A
VERIFICAR
			Signature verificadsa =
Signature.getInstance("SHAlwithDSA");
			verificadsa.initVerify(clavepub);
			verificadsa.update(mensaje.getBytes());
			boolean check = verificadsa.verify(firma);
			if(check)
				System.out.println("FIRMA VERIFICADA CON CLAVE
PÚBLICA") ;
			else
				System.out.println("FIRMA NO VERIFICADA");

		} catch (NoSuchAlgorithmException e1) {
e1.printStackTrace();
		} catch (InvalidKeyException e) { e.printStackTrace();
		} catch (SignatureException e) { e.printStackTrace(); }

	}//Fin de main
}//Fin de Ejemplo7
```

Almacenar las claves pública y privada en ficheros

Para almacenar la clave privada en disco es necesario codificarla en formato PKCS8 usando la clase **PKCS8EncodedKeySpec**.

MÉTODOS	MISIÓN
PKCS8EncodedKeySpec (byte [] encodedKey)	Crea un nuevo objeto *PKCS8EncodedKeySpec* con la clave codificada
byte [] getEncoded ()	Devuelve los bytes codificados de la clave de acuerdo con el estándar PKCS # 8
String getFormat ()	Devuelve el nombre del formato de codificación asociado con esta especificación de clave

```
PKCS8EncodedKeySpec pk8Spec =
            new PKCS8EncodedKeySpec(clavepriv.getEncoded());
            //Escribir a fichero binario la clave privada
FileOutputStream outpriv =
new FileOutputStream("Clave.privada");
outpriv.write(pk8Spec.getEncoded());
outpriv.close();
```

Para almacenar la clave pública en disco es necesario codificarla en formato X.509 usando la clase **X509EncodedKeySpec.**

MÉTODOS	MISIÓN
X509EncodedKeySpec(byte[] encodedKey)	Crea un nuevo objeto *X509EncodedKeySpec* con la clave codificada
byte [] getEncoded ()	Devuelve los bytes codificados de la clave de acuerdo con el estándar X.509
String getFormat ()	Devuelve el nombre del formato de codificación asociado con esta especificación de clave

```
X509EncodedKeySpec pkX509 =
            new X509EncodedKeySpec(clavepub.getEncoded());
            //Escribir a fichero binario la clave pública
FileOutputStream outpub =
new FileOutputStream("Clave.publica");
outpub.write(pkX509.getEncoded());
outpub.close();
```

Recuperar las claves pública y privada de ficheros

Clase **KeyFactory**. Para recuperar las claves de los ficheros que proporciona métodos para convertir claves de formato criptográfico (PKCS8, X.509) a especificaciones de claves y viceversa. Su constructor y alguno de sus métodos:

MÉTODOS	MISIÓN
static KeyFactory getInstance(String algoritmo)	Devuelve un objeto **KeyFactory** capaz de importar y exportar las claves generadas con el algoritmo dado. Puede lanzar la excepción *NoSuchAlgorithmException*
static KeyFactory getInstance(String algoritmo, String provider)	En el segundo método se especifica el proveedor. Si el nombre de proveedor no se encuentra se produce *NoSuchProviderException*
PrivateKey generatePrivate (KeySpec KeySpec)	Genera un objeto de clave privada a partir de la especificación de clave suministrada
PublicKey generatePublic (KeySpec KeySpec)	Genera un objeto de clave pública a partir de la especificación de clave suministrada

Firmar los datos de un fichero con la clave privada

Archivo EJEMPLO8.JAVA

```
import java.io.*;
import java.security.*;
import java.security.spec.*;

public class Ejemplo8 {

	public static void main(String[] args) {

		try {
			// LECTURA DEL FICHERO DE CLAVE PRIVADA
			FileInputStream inpriv = new
FileInputStream("Clave.privada");
			byte[] bufferPriv = new byte[inpriv.available()];
			inpriv.read(bufferPriv);// lectura de bytes
			inpriv.close();

			//RECUPERA CLAVE PRIVADA DESDE DATOS CODIFICADOS EN
FORMATO PKCS8
			PKCS8EncodedKeySpec clavePrivadaSpec = new
PKCS8EncodedKeySpec(bufferPriv);
			KeyFactory keyDSA = KeyFactory.getInstance("DSA");
			PrivateKey clavePrivada =
keyDSA.generatePrivate(clavePrivadaSpec);

			//INICIALIZA FIRMA CON CLAVE PRIVADA
			Signature dsa =
Signature.getInstance("SHA1withDSA");
			dsa.initSign (clavePrivada);

			//LECTURA DEL FICHERO A FIRMAR
			//Se suministra al objeto Signature los datos a
firmar
			FileInputStream fichero = new
FileInputStream("FICHERO.DAT");
			BufferedInputStream bis = new
BufferedInputStream(fichero);
			byte[] buffer = new byte[bis.available()];
			int len;
			while ((len = bis.read(buffer)) >= 0)
				dsa.update(buffer, 0, len);
			bis.close();

			//GENERA LA FIRMA DE LOS DATOS DEL FICHERO
			byte[] firma = dsa.sign();

			// GUARDA LA FIRMA EN OTRO FICHERO
			FileOutputStream fos = new
FileOutputStream("FICHERO.FIRMA");
			fos.write(firma);
			fos.close();

} catch (Exception e1) { e1.printStackTrace();
} }
```

Genera la firma del fichero *DATOS.DAT* a partir de la clave privada almacenada en el fichero *Clave.privada*. La firma se almacenará en el fichero *DATOS.FIRMA*.

Verificar la firma de un fichero con la clave pública

Archivo EJEMPLO9.JAVA

```
import java.io.*;
import java.security.*;
import java.security.spec.*;
public class Ejemplo9 {
 public static void main(String[] args) {
   try {
         //LECTURA DE LA CLAVE PUBLICA DEL FICHERO
         FileInputStream inpub = new FileInputStream("Clave.publica");
         byte[] bufferPub = new byte[inpub.available()];
         inpub.read(bufferPub);// lectura de bytes
         inpub.close();

         //RECUPERA k PUBLIC DESDE DATOS CODIFI EN FORMATO X509
         KeyFactory keyDSA = KeyFactory.getInstance("DSA");
         X509EncodedKeySpec clavePublicaSpec = new
X509EncodedKeySpec(bufferPub);
         PublicKey clavePublica =
keyDSA.generatePublic(clavePublicaSpec);

         //LECTURA DEL FICHERO QUE CONTIENE LA FIRMA
         FileInputStream firmafic = new
FileInputStream("FICHERO.FIRMA");
         byte[] firma = new byte[firmafic.available()];
         firmafic.read(firma); firmafic.close();

         //INIC EL OBJETO Signature CON k PÚBLICA PARA VERIFICAR
         Signature dsa = Signature.getInstance("SHAlwithDSA");
         dsa.initVerify (clavePublica);

         //LECTURA DEL FICHERO QUE CONTIENE LOS DATOS A VERIFICAR
         //Se dá al objeto Signature los datos a verificar
         FileInputStream fichero = new FileInputStream("FICHERO.DAT");
         BufferedInputStream bis = new BufferedInputStream(fichero);
         byte[] buffer = new byte[bis.available()];
         int len;
         while ((len = bis.read(buffer)) >= 0)
             dsa.update(buffer, 0, len);
         bis.close();

         //VERIFICAR LA FIRMA DE LOS DATOS LEIDOS
         boolean verifica = dsa.verify(firma);
         //COMPROBAR LA VERIFICACIÓN
         if (verifica)
             System.out.println("DATOS OK CON SU FIRMA.");
         else
             System.out.println("DATOS KO CON SU FIRMA");
   } catch (Exception el) { el.printStackTrace(); }
 }//Fin de main
}//Fin de Ejemplo9
```

Necesitamos la clave púlica almacenada en el fichero *Clave. Pública*, la firma del fichero almacenada en *DATOS.FIRMA* y el fichero de datos *DATOS.DAT*. En primer lugar, obtendremos la clave pública del fichero *Clave. Pública*, a continuación, obtenemos la firma digital almacenada en el fichero *DATOS.FIRMA*. A continuación, se leen los datos del fichero de datos *DATOS.DAT* y se suministran al objeto **Signature**. Por último, se verifica la firma con la clave pública.

Herramientas para firmar ficheros

Java dispone de la herramienta de línea de comandos **keytool** para generar y manipular certificados.

Para firmar un documento seguiremos los siguientes pasos:

1. Crear un fichero JAR que contiene el documento a firmar.

```
jar cvf Contrato.jar Contrato.pdf
```

2. Generar las claves pública y privada (si no existen), con keytool-genkey.

```
keytool –genkey –alias FirmaContrato –keystore
AlmacenClaves
```

Creamos un almacén de claves (**keystore**) con el nombre *AlmacenClaves*.
FirmaContrato es el nombre con el que haremos referencia al par de claves creado.

Nos pedirá contraseña para el almacén de claves y para la clave privada del par de claves generado.

El certificado generado tiene una validez de 90 días a no ser que se especifique la opción **`-validity`** en **keytool**.
Los certificados autofirmados son útiles para desarrollar y probar una aplicación.
La aplicación está firmada con un certificado que no es de confianza, por tanto, al ejecutarla nos preguntará antes si queremos ejecutarla.
Se recomienda no importar en un almacén de claves un certificado en el que no se confíe plenamente.

3. Firmar el fichero JAR, usando jarsigner y la clave privada.

```
jarsigner –keystore AlmacenClaves –signedjar
DocumentoFirmado.jar Contrato.jar FirmaContrato
```

4. Con keytool –export exportar el certificado de clave pública para que el receptor autentique la firma del emisor.

```
keytool –export –keystore AlmacenClaves –alias
FirmaContrato –file MariaJesus.cer
```

5. Por último suministrar el fichero JAR firmado y el certificado al receptor.

El receptor necesita importar el certificado como un certificado de confianza

```
keytool –import –alias MJesus –file MariaJesus.cer –
keystore AlmacenReceptor
```

y verificar la firma del fichero JAR.

```
jarsigner –verify –verbose –certs –keystore AlmacenReceptor
DocumentoFirmado.jar
```

Clase Cipher

Para crear un objeto **Cipher** se llama al método `getInstance()` pasando como argumento el algoritmo y opcionalmente, el nombre de un proveedor.

MÉTODOS	MISIÓN
static Cipher getInstance(String algoritmo) static Cipher getInstance(String algoritmo, String proveedor)	Devuelve un objeto *Cipher* que implementa el algoritmo especificado. En el segundo caso se especifica el proveedor El algoritmo tiene la forma: algoritmo/modo/relleno o algoritmo. Por ejemplo: AES/CBC/NoPadding, AES/CBC/PKCS5Padding, etc. Puede lanzar la excepción *NoSuchAlgorithmException* y *NoSuchPaddingException*; si el nombre de proveedor no se encuentra se produce *NoSuchProviderException*
int getBlockSize()	Devuelve el tamaño del bloque en bytes
int getOutputSize(int inputLen)	Devuelve el tamaño en bytes de un búfer de salida que es necesario si la siguiente entrada tiene el número de bytes indicado
void init(int modo, Key clave)	Inicializa el objeto del algoritmo codificador, *modo* puede ser ENCRYPT_MODE (encriptar datos), DECRYPT_MODE (desencriptar datos), WRAP_MODE o UNWRAP_MODE. Los modos *wrap* y *unwrap* se utilizan para encriptar una clave con otra Para inicializar el objeto hay que proporcionar una clave
byte[] update(byte[] entrada) byte[] update(byte[] entrada, int desplazamiento, int longitud)	Transforma (encripta o desencripta) el bloque de datos indicado en *entrada*
byte[] doFinal() byte[] doFinal(byte[] entrada)	Termina la operación de cifrado o descifrado y limpia el buffer de ese objeto algoritmo. En el segundo método se indican los bytes de entrada que se procesarán
byte[] wrap(Key key)	Envuelve una clave. Este método y el siguiente se utilizan para encriptar y desencriptar una clave a partir de otra
Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType)	Desenvuelve una clave previamente envuelta. *wrappedKey* es la clave a desenvolver, *wrappedKeyAlgorithm* es el algoritmo asociado con la clave envuelta, *wrappedKeyType* es el tipo de la clave envuelta; debe ser uno de los siguientes: SECRET_KEY, PRIVATE_KEY o PUBLIC_KEY

Como algoritmo en el método `getInstance()` se pueden poner los siguientes (**algoritmo/modo/relleno**), entre paréntesis se especifica el tamañi de la clave en bits:

AES/CBC/NoPadding (128)
AES/CBC/PKCS5Padding (128)
AES/ECB/NoPadding (128)
AES/ECB/PKCS5Padding (128)
DES/CBC/NoPadding (56)
DES/CBC/PKCS5Padding (56)
DES/ECB/NoPadding (56)
DES/ECB/PKCS5Padding (56)
DESede/CBC/NoPadding (168)
DESede/CBC/PKCS5Padding (168)
DESede/ECB/NoPadding (168)
DESede/ECB/PKCS5Padding (168)
RSA/ECB/PKCS1Padding (1024, 2048)
RSA/ECB/OAEPWithSHA-1AndMGF1Padding (1024, 2048)
RSA/ECB/OAEPWithSHA-256AndMGF1Padding (1024, 2048)

Los modos son la forma de trabajar del algoritmo

- **ECB** (Electronic Cookbook Mode). Los mensajes se dividen en bloques y cada uno de ellos es cifrado por separado por separado utilizando la misma clave K. A bloques de texto plano o claro idénticos les corresponden bloques idénticos de texto cifrado, de anera que se pueden reconocer estos patrones. De ahí que no sea recomendable.
- **CBC** (Cipher Block Chaining), a cada bloque de texto plano se le aplica la operación XOR con el bloque cifrado anterior antes de ser cifrado. De esta forma, cada bloque de texto cifrado depende de todo el texto en claro procesado hasta este punto. Para hacer cada men saje único se utiliza asimismo un vector de inicialización.

El relleno se utiliza cuando el mensaje a cifrar no es múltiplo de la longitud de cifrado del algoritmo, entonces es necesario indicar la forma de rellenar los últimos bloques.

Clase KeyGenerator

Proporciona funcionalidades para generar claves secretas para usarse en algoritmos simétricos. Algunos métodos son:

MÉTODOS	MISIÓN
static KeyGenerator getInstance(String algoritmo)	Devuelve un objeto *KeyGenerator* que genera claves secretas para el algoritmo especificado, por ejemplo "DES".Puede lanzar la excepción *NoSuchAlgorithmException*
void init(int keysize, SecureRandom random) void init(int keysize) void init(SecureRandom random)	Inicializa el generador de claves para un determinado tamaño de clave y un generador de números aleatorios En el segundo método solo se proporciona el tamaño de clave y en el tercero el generador de números aleatorios
SecretKey generateKey()	Genera una clave secreta

Pasos para encriptar y desencriptar con clave secreta

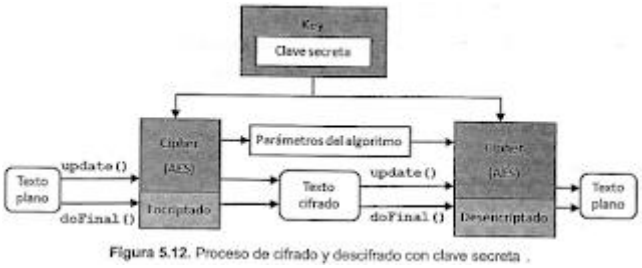


Figura 5.12. Proceso de cifrado y descifrado con clave secreta .

➔ Creamos la clave secreta usando **AES** o **DES.**
➔ Creamos un objeto **Cipher**con el **algoritmo/modo/relleno** que creamos oportuno, lo inicializamos en **modo encriptación** con la clave creada anteriormente.
➔ Realizamos el cifrado de la información con el método `doFinal()`.
➔ Configuramos el objeto **Cipher** en modo desencriptación con la clave anterior para desencriptar el texto, usamos el método `doFinal()`.

Archivo EJEMPLO10.JAVA

```
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.Key;
import java.security.NoSuchAlgorithmException;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;

public class Ejemplo10 {

	public static void main(String[] args) {


		try {

			//Creamos la clave secreta usando el algoritmo AES y
definimos un tamaño de clave de 128 bits
			KeyGenerator kg = KeyGenerator.getInstance("AES");
			kg.init (128);
			SecretKey clave = kg.generateKey();

			//Creamos un objeto Cipher con el algoritmo
AES/ECB/PKCS5Padding, lo inicializamos en modo encriptación con la
clave creada anteriormente.
			Cipher c =
Cipher.getInstance("AES/ECB/PKCS5Padding");
			c.init(Cipher.ENCRYPT_MODE, clave);

			//Realizamos el cifrado de la información con el
método doFinal()
			byte textoPlano[] = "Esto es un Texto
Plano".getBytes();
			byte textoCifrado[] = c.doFinal(textoPlano);
			System.out.println("Encriptado: "+ new
String(textoCifrado));

			//Configuramos el objeto Cipher en modo
desencriptación con la clave anterior para desencriptar el texto,
usamos el método doFinal()
			c.init(Cipher.DECRYPT_MODE, clave);
			byte desencriptado[] = c.doFinal(textoCifrado);
			System.out.println("Desencriptado: "+ new
String(desencriptado));


			/*
			//Muchos modos de algoritmo (por ejemplo CBC)
requieren un vector de inicialización que se especifica cuando se
inicializa
			//el objeto Cipher en modo desencriptación. En estos
casos, se debe pasar al método init() el vector de inicialización.
```

```
                  //La clase IvParameterSpec se usa para hacer esto en
el cifrado DES.
                  KeyGenerator kg = KeyGenerator.getInstance("DES");
                  Cipher c =
Cipher.getInstance("DES/CBC/PKCS5Padding");
                  Key clave = kg.generateKey();

                  //Devuelve el vector IV inicializado en un nuevo
buffer
                  byte iv[]=c.getIV();
                  IvParameterSpec dps = new IvParameterSpec(iv);
                  c.init(Cipher.DECRYPT_MODE, clave, dps);
                  */

            } catch (NoSuchAlgorithmException e) {
            } catch (NoSuchPaddingException e) {
            } catch (InvalidKeyException e) {
            } catch (IllegalBlockSizeException e) {
            } catch (BadPaddingException e) {
      //    } catch (InvalidAlgorithmParameterException e) {
            }

      }// Fin de main
}// Fin de Ejemplo10
```

Almacenar la clave secreta en un fichero

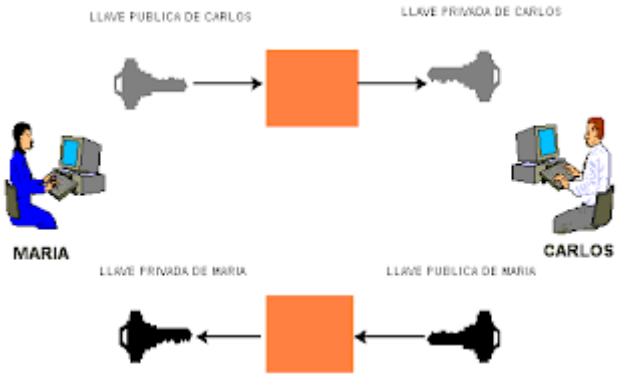
```
import java.io.*;
import java.security.*;

import javax.crypto.*;

public class AlmacenaClaveSecreta {

      public static void main(String[] args) {

            try {
                  KeyGenerator kg = KeyGenerator.getInstance("AES");
                  kg.init(128);
                  //genera clave secreta
                  SecretKey clave = kg.generateKey();
                  ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("Clave.secreta"));
                  out.writeObject(clave);
                  out.close();

                  /*
                  //Para recuperar la clave secreta del fichero
                  ObjectInputStream in = new ObjectInputStream(new
FileInputStream("Clave.secreta"));
                  Key secreta = (Key) in.readObject();
                  in.close();
                  */

            } catch (NoSuchAlgorithmException e) {e.printStackTrace();
            } catch (FileNotFoundException e) {e.printStackTrace();
            //} catch (ClassNotFoundException e) {
e.printStackTrace();//Para recuperar la clave secreta
            } catch (IOException e) {e.printStackTrace();}
```

```
      }//Fin de main
}//Fin de AlmacenaClaveSecreta
```

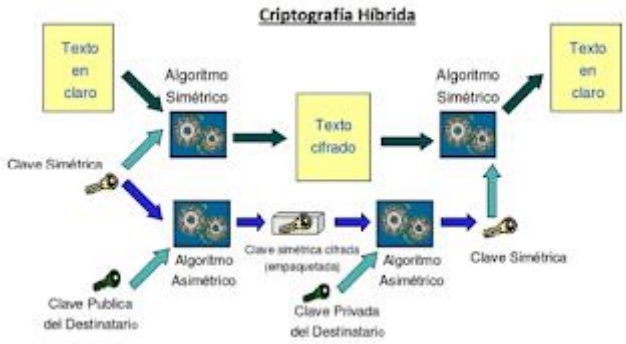
Genera una clave secreta **AES** y la almacena en el fichero **Clave.secreta**.

Encriptar y desencriptar con clave pública

Conversación encriptada



Criptografía híbrida



Archivo EJEMPLO12.JAVA

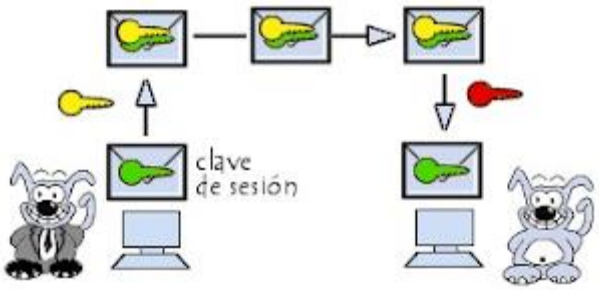
```
import java.security.*;
import javax.crypto.*;

public class Ejemplo12 {

      public static void main(String args[]) {

            try {
                  //SE CREA EL PAR DE CLAVES PÚBLICA Y PRIVADA
                  KeyPairGenerator keyGen =
KeyPairGenerator.getInstance("RSA");
                  keyGen.initialize (1024);
                  KeyPair par = keyGen.generateKeyPair();
                  PrivateKey clavepriv = par.getPrivate();
                  PublicKey clavepub = par.getPublic();
```

```
                    //SE CREA LA CLAVE SECRETA AES
                    KeyGenerator kg = KeyGenerator.getInstance("AES");
                    kg.init (128);
                    SecretKey clavesecreta = kg.generateKey();

                    //SE ENCRIPTA LA CLAVE SECRETA CON LA CLAVE RSA
PÚBLICA
                    Cipher c =
Cipher.getInstance("RSA/ECB/PKCS1Padding");
                    c.init(Cipher.WRAP_MODE, clavepub);
                    byte claveenvuelta[] = c.wrap(clavesecreta);

                    //CIFRAMOS TEXTO CON LA CLAVE SECRETA
                    c = Cipher.getInstance("AES/ECB/PKCS5Padding");
                    c.init(Cipher.ENCRYPT_MODE, clavesecreta);
                    byte textoPlano[] = "Esto es un Texto
Plano".getBytes();
                    byte textoCifrado[] = c.doFinal(textoPlano);
                    System.out.println("Encriptado: " + new
String(textoCifrado));

            /* Para desencriptar el texto primero necesitamos
desencriptar la clave Secreta con la clave privada y a continuación
desencriptar el texto con esa clave; usaremos el método unwrap():*/

                    //SE DESENCRIPTA LA CLAVE SECRETA CON LA CLAVE RSA
PRIVADA
                    Cipher c2 =
Cipher.getInstance("RSA/ECB/PKCS1Padding");
                    c2.init(Cipher.UNWRAP_MODE, clavepriv);
                    Key clavedesenvuelta= c2.unwrap (claveenvuelta,
"AES", Cipher.SECRET_KEY);

                    //DESCIFRAMOS EL TEXTO CON LA CLAVE DESENVUELTA
                    c2 = Cipher.getInstance("AES/ECB/PKCS5Padding");
                    c2.init(Cipher.DECRYPT_MODE, clavedesenvuelta);
                    byte desencriptado[] = c2.doFinal(textoCifrado);
                    System.out.println("DesenCriptado:" + new
String(desencriptado));

            } catch (Exception e) { e.printStackTrace(); }

      }//Fin de main
}//Fin de Ejemplo12
```

- Se genera un par de claves pública y privada con el algoritmo RSA.
- Se crea una clave secreta con el algoritmo AES.
- Esta clave se creará para encriptar el texto.
- La clave secreta es encriptada mediante la clave pública utilizando el método `wrap()`
- Para desencriptar el texto primero necesitamos desencriptar la clave Secreta con la clave priada y a continuación desencriptar el texto con esa clave; usaremos el método `unwrap()`

Clave de sesión

Es un término medio entre el cifrado simétrico y asimétrico que permite combinar las dos técnicas. Consiste en generar una clave de sesión K y cifrarla usando la clave pública del receptor. El receptor descifra la clave de sesión usando su clave privada. El emisor y el receptor comparten una clave que solo ellos conocen y pueden cifrar sus comunicaciones usando la misma clave de sesión.



Encriptar y desencriptar flujos de datos

Clase CipherOutputStream. Encriptar datos hacia un fichero

MÉTODOS	MISIÓN
CipherOutputStream(OutputStream salida, Cipher codificador)	Construye un flujo de salida que escribe datos en *salida* y los encripta o desencripta utilizando el objeto *Cipher* indicado
void write(int b)	Escribe el byte especificado en el stream de salida
void write(byte[] b, int off, int len)	Escribe *len* bytes en el array de bytes especificado comenzando en *off*
void flush()	Limpia el buffer del objeto *Cipher* y efectúa el relleno si es necesario

CipherInputStream. Leer y desencriptar datos de un fichero

MÉTODOS	MISIÓN
CipherInputStream(InputStream entrada, Cipher codificador)	Construye un flujo de entrada que lee datos procedentes de *entrada* y los desencripta o encripta utilizando el objeto *Cipher* indicado
int read()	Lee el siguiente byte de datos del flujo de entrada
int read(byte[] b, int off, int len)	Lee hasta *len* bytes de datos de este flujo de entrada en una matriz de bytes

Ambas manipular de forma transparente las llamadas a `update()` y `doFinal()`

Archivo EJEMPLO13CIFRA.JAVA

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

public class Ejemplo13Cifra {

	public static void main(String[] args) {

		try {
			//RECUPERAMOS CLAVE SECRETA DEL FICHERO
			ObjectInputStream oin = new ObjectInputStream( new
FileInputStream("Clave.secreta"));
			Key clavesecreta = (Key) oin.readObject();
			oin.close();
```

```
                  //SE DEFINE EL OBJETO Cipher para encriptar
                  Cipher c =
Cipher.getInstance("AES/ECB/PKCS5Padding");
                  c.init(Cipher.ENCRYPT_MODE, clavesecreta);

                  //FICHERO A CIFRAR
                  FileInputStream filein = new
FileInputStream("FICHERO.pdf");
                  //OBJETO CipherOutputStream donde se almacena el
fichero cifrado
                  CipherOutputStream out = new CipherOutputStream( new
FileOutputStream("FicheroPDF.Cifrado"), c);
                  int tambloque = c.getBlockSize();//tamaño de bloque
objeto Cipher
                  byte[] bytes = new byte[tambloque];//bloque de bytes

                  //LEEMOS BLOQUES DE BYTES DEL FICHERO PDF
                  //Y int LO VAMOS ESCRIBIENDO AL CipherOutputStream
                  int i = filein.read(bytes);
                  while (i != -1) {
                        out.write(bytes, 0, i);
                        i = filein.read(bytes);
                  }

                  out.flush();
                  out.close();
                  filein.close();
                  System.out.println("Fichero cifrado con clave
secreta.");

            } catch (Exception e) {e.printStackTrace();}

      }//Fin de main
}// Fin de Ejemplo13Cifra
```

Utiliza la clave secreta almacenada en un fichero llamado para cifrar un documento PDF de nombre *Fichero.pdf*.

Archivo EJEMPLO13DESCIFRA.JAVA

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

public class Ejemplo13Descifra {

      public static void main(String[] args) {

            try {
                  //RECUPERAMOS CLAVE SECRETA DEL FICHERO
                  ObjectInputStream oin = new ObjectInputStream(new
FileInputStream("Clave.secreta"));
                  Key clavesecreta = (Key) oin.readObject();
                  oin.close();

                  //SE DEFINE EL OBJETO Cipher para desencriptar
                  Cipher c =
Cipher.getInstance("AES/ECB/PKCS5Padding");
                  c.init(Cipher.DECRYPT_MODE, clavesecreta);
```

```
                  //OBJETO CipherInputStream CUYO CONTENIDO SE VA A
DESCIFRAR
                  CipherInputStream in = new CipherInputStream(new
FileInputStream("FicheroPDF.Cifrado"), c);
                  int tambloque = c.getBlockSize();//tamaño de bloque
                  byte[] bytes = new byte[tambloque];//bloque de bytes

                  //FICHERO CON EL CONTENIDO DESCIFRADO QUE SE CREARÁ
                  FileOutputStream fileout = new
FileOutputStream("FICHERODescifrado.pdf");

                  //LEEMOS BLOQUES DE BYTES DEL FICHERO cifrado
                  //Y LO VAMOS ESCRIBIENDO desencriptados al
FileOutputStream
                  int i = in.read(bytes);
                  while (i != -1){
                        fileout.write(bytes, 0, i);
                        i = in.read(bytes);
                  }
                  fileout.close();
                  in.close();
                  System.out.println("Fichero descifrado con clave
secreta.");

            } catch (Exception e) {e.printStackTrace();}

      }//Fin de main
}//Fin de Ejemplo13Descifra
```

Utiliza la clase **CipherInputStream** para leer y desencriptar datos de un fichero cifrado.