

# Tutorial de JCA (Java Cryptographic Architecture)

SSI 2021/22

13 de septiembre de 2021

## Índice

<b>1. JCA: Java Cryptographics Architecture</b>	<b>1</b>
1.1. Estructura . . . . .	2
1.2. Documentación adicional . . . . .	2
<b>2. Provider BouncyCastle</b>	<b>2</b>
2.1. Descarga y documentación . . . . .	2
2.2. Instalación y uso . . . . .	2
<b>3. Ejemplos</b>	<b>3</b>
3.1. Funciones hash criptográficas . . . . .	3
3.2. Cifrado simétrico . . . . .	4
3.3. Cifrado asimétrico . . . . .	5
3.3.1. Creación de claves asimétricas . . . . .	5
3.4. Almacenamiento de claves . . . . .	6
3.5. Firmas digitales . . . . .	6
3.5.1. Generación de firmas . . . . .	7
3.5.2. Verificación de firmas . . . . .	7

## 1. JCA: Java Cryptographics Architecture

Framework para criptografía que forma parte de la distribución estándar de la JVM (máquina virtual de Java). Reemplaza (y amplía) al API JCE (*Java Cryptographic Extensions*)

Ofrece un API (*application programming interface*) que permite:

- generación de claves (claves secretas y pares de claves pública y privada)
- cifrado simétrico (DES, 3DES, IDEA, etc)
- cifrado asimétrico (RSA, DSA, Diffie-Hellman, ElGamal...)
- funciones de resumen (MD5 y SHA1 ) y algoritmos MAC (*Message Authentication Code*)
- generación y validación de firmas
- acuerdo de claves

## 1.1. Estructura

Las implementaciones de los distintos algoritmos de cifrado, generación de claves y demás son ofertadas por paquetes externos denominados *providers*.

- La distribución básica de Java incluye por defecto el provider "SUN" con implementaciones de los algoritmos más representativos.
- Otros fabricantes ofrecen providers adicionales que incluyen nuevos algoritmos o implementaciones alternativas de los ya existentes en el provider "SUN".

Así, es posible dotar a JCA de nuevas funcionalidades sin necesidad de cambiar el API básica y permitir la distribución de algoritmos criptográficos con limitaciones de exportación.

Para usar las clases y métodos del API JCA las aplicaciones tienen que importar, como mínimo, los siguientes paquetes:

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

import javax.crypto.*;
import javax.crypto.interfaces.*;
import javax.crypto.spec.*;
```

## 1.2. Documentación adicional

- Descripción infraestructura de seguridad en Java SE
- Descripción de JCA (Java Cryptographic API)
- Listado con los nombres estándar de los algoritmos en JCA/JCE
- Javadoc JCA:
  - paquete java.security
  - paquete javax.crypto
- Manual de cifrado con JCE (en español)

## 2. Provider BouncyCastle

Bouncy Castle es un proyecto de software libre que pretende desarrollar una serie de librerías criptográficas libres y, entre otros, ofrece un provider para el JCA de Java.

### 2.1. Descarga y documentación

En la página web del proyecto (<http://www.bouncycastle.org>) es posible descargar la versión actual (1.69) del provider Bouncy Castle para distintas versiones de la máquina virtual de Java.

También incluye información resumida y el javadoc completo de la distribución.

### 2.2. Instalación y uso

Para usar las clases de JCA/JCE ofrecidas por BouncyCastle, basta con incluir en el código que lo utilice la siguiente orden de importación:

```
import org.bouncycastle.jce.provider.BouncyCastleProvider;
```

Dentro del código será necesario indicar la carga del provider BouncyCastle del siguiente modo:

```
Security.addProvider(new BouncyCastleProvider());
```

Este código se deberá situar antes de cualquier uso que se realice de este provider. La abreviatura que identifica a este provider es "BC" y deberá indicarse cuando se solicite alguna de las implementaciones de algoritmos que ofrece el provider BouncyCastle.

```
Cipher cifrador = Cipher.getInstance("DES/ECB/PKCS1Padding", "BC");
KeyGenerator keyGen = KeyGenerator.getInstance("AES", "BC");
```

Para compilar y ejecutar estos programas será necesario indicar mediante el parámetro `-classpath` el fichero `.jar` con las clases BouncyCastle adecuado a la máquina virtual Java que esté instalada.

```
$ javac -classpath ".:bcprov-jdk15on-xxx.jar" Fichero.java
$ java -classpath ".:bcprov-jdk15on-xxx.jar" Fichero
```

```
C:\ javac -classpath ".;bcprov-jdk15on-xxx.jar" Fichero.java
C:\ java -classpath ".;bcprov-jdk15on-xxx.jar" Fichero
```

Otra alternativa que evita especificar el CLASSPATH en cada invocación es copiar el fichero JAR de BouncyCastle en el directorio de librerías externas de la instalación del JDK o JRE, de modo que se incluya en el CLASSPATH por defecto de la máquina virtual Java.

```
$ cp bcbprov-jdk15on-xxx.jar /usr/lib/jvm/java-x-yyy/jre/lib/ext/
```

## 3. Ejemplos

### 3.1. Funciones hash criptográficas

El acceso al uso de funciones Hash criptográficas se lleva a cabo mediante instancias que hereden de `MessageDigest`

**Creación de instancias** Realizada mediante un método factoría (no con `new()`). Será el provider quien cree el objeto concreto utilizando su propia implementación del correspondiente algoritmo Hash.

Se debe indicar el nombre (alias) del algoritmo de HASH y opcionalmente el nombre del provider.

```
static MessageDigest getInstance(String algorithm)
static MessageDigest getInstance(String algorithm, String provider)
```

#### Cálculo de resúmenes

- Se debe "alimentar" al algoritmo con los datos a resumir (mediante los métodos `update()`)

```
void update(byte input)
void update(byte[] input)
void update(byte[] input, int offset, int len)
```

- Se obtiene el resumen invocando al método `digest()` que finaliza las operaciones

```
byte[] digest()
byte[] digest(byte[] input) // Método de conveniencia, crea el resumen de los datos directam
```

Ejemplo de uso del algoritmo MD5: EjemploHash

```
$ javac EjemploHash.java
$ java EjemploHash fichero.txt
```

### 3.2. Cifrado simétrico

El uso de cifradores simétricos (y también asimétricos) se hace mediante instancias que heredan de la clase `Cipher`

**Creación de instancias** Realizada mediante un método factoría (no con `new()`). Será el provider quien cree el objeto concreto utilizando su implementación del correspondiente algoritmo.

Se debe indicar una especificación del esquema de cifrado a utilizar.

- Cómo mínimo esta especificación incluirá el nombre (alias) del algoritmo de cifrado.
- En los casos en los que sea necesario (típicamente con cifradores de bloque) se incluirá además la especificación del modo de funcionamiento (ECB, CBC, ...) y del algoritmo de relleno.

(Opcionalmente puede indicarse el nombre del provider para seleccionar la implementación concreta a utilizar)

```
static Cipher getInstance(String transformation)
static Cipher getInstance(String transformation, String provider)
```

**Creación de la clave** Las claves se gestionan mediante objetos que implementan la interfaz `SecretKey` (que a su vez hereda de la interfaz `Key`)

Las claves se crean empleando un objeto `KeyGenerator`.

1. Las instancias del `KeyGenerator` se obtienen mediante llamadas a un método factoría, en las que se debe indicar el alias del algoritmo de cifrado con el que se utilizará la clave y, opcionalmente, el nombre del provider.

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

2. Configuración del `KeyGenerator` (normalmente especificación del tamaño de clave)

```
generadorDES.init(56); // clave de 56 bits
```

3. Creación de la clave

```
SecretKey clave = generadorDES.generateKey();
```

**Cifrado y descifrado** 1. Se debe establecer el *modo de funcionamiento* del cifrador (método `init(...)`)

- `DECRYPT_MODE` : modo descifrado
- `ENCRYPT_MODE` : modo cifrado
- `PRIVATE_KEY` : clave para indicar el "desempaquetado" de una clave privada
- `PUBLIC_KEY` : clave para indicar el "desempaquetado" de una clave pública
- `SECRET_KEY` : clave para indicar el "desempaquetado" de una clave secreta (simétrica)
- `UNWRAP_MODE` : modo "desempaquetado" de claves (descifrado de claves)
- `WRAP_MODE` : modo "empaquetado" de claves (cifrado de claves)

```
cifrador.init(Cipher.ENCRYPT_MODE, clave);
```

```
cifrador.init(Cipher.DECRYPT_MODE, clave);
```

2. Se debe "alimentar" al algoritmo con los datos a cifrar/descifrar (mediante los métodos `update()`)

```
byte[] update(byte[] input)
byte[] update(byte[] input, int offset, int len)
```

Devuelve un array de `byte` con el resultado "parcial" del cifrado/descifrado

3. Se debe finalizar el cifrado/descifrado invocando al método `doFinal()` que finaliza las operaciones (**Importante:** SIEMPRE es necesario invocarlo por si hay que manejar relleno)

```
byte[] doFinal()
byte[] doFinal(byte[] input) // Metodo de conveniencia, com
byte[] doFinal(byte[] input, int inputOffset, int inputLen) // Metodo de conveniencia, com
```

Devuelve un array de `byte` con el resultado del cifrado/descifrado del último bloque

Ejemplo de uso del algoritmo DES: EjemploDES

```
$ javac EjemploDES.java
$ java EjemploDES fichero.txt
```

### 3.3. Cifrado asimétrico

El funcionamiento de los cifradores asimétricos es idéntico al de los simétricos ya que también hacen uso de objetos que hereden de la clase `Cypher`.

La única diferencia es el modo en que se generan los pares de claves públicas y privadas y el uso que se hace de ellas (en cifrado y/o descifrado).

#### 3.3.1. Creación de claves asimétricas

Las claves se gestionan mediante objetos que implementan el interfaz `PublicKey` y `PrivateKey` (que a su vez hereda del interfaz `Key`)

Las claves se crean empleando un objeto `KeyPairGenerator` específico para cada algoritmo asimétrico.

1. Creación del `KeyPairGenerator` empleando un método factoría

Se debe indicar el alias del algoritmo de cifrado y el nombre del provider [inicialmente el provider por defecto "SUN" no incluía RSA por limitaciones a la exportación de algoritmos de cifrado].

```
Security.addProvider(new BouncyCastleProvider()); // Cargar el provider BC
...
KeyPairGenerator keyGenRSA = KeyPairGenerator.getInstance("RSA", "BC"); // Usa BouncyCastle
```

2. Configuración del `KeyPairGenerator` (normalmente especificación del tamaño de clave)

```
keyGenRSA.initialize(1024); // clave RSA de 1024 bits
```

3. Creación del par de claves (y recuperación de las claves pública y privada)

```
KeyPair clavesRSA = keyGenRSA.generateKeyPair();

PrivateKey clavePrivada = clavesRSA.getPrivate();
PublicKey clavePublica = clavesRSA.getPublic();
```

**Nota:** Se pueden inspeccionar los parámetros de las claves pública y privada RSA forzando un *cast* a los interfaces `RSAPrivateKey` y `RSAPublicKey` (se necesita hacer un `import` del paquete `java.security.interfaces.*`)

```
RSAPrivateKey clavePrivadaRSA = (RSAPrivateKey) clavePrivada;
System.out.println("exponente descifrado: " + clavePrivadaRSA.getPrivateExponent().toString());
System.out.println("modulo: " + clavePrivadaRSA.getModulus().toString());

RSAPublicKey clavePublicaRSA = (RSAPublicKey) clavePublica;
System.out.println("exponente cifrado: " + clavePublicaRSA.getPublicExponent().toString());
System.out.println("modulo: " + clavePublicaRSA.getModulus().toString());
```

Ejemplo de uso del algoritmo RSA: EjemploRSA

```
$ javac -classpath ".:bcprov-jdk15on-xxx.jar" EjemploRSA.java (en Linux)
$ java -classpath ".:bcprov-jdk15on-xxx.jar" EjemploRSA (en Linux)
```

### 3.4. Almacenamiento de claves

Dado que JCE tiene una arquitectura basada en *providers* donde las clases de implementación del *provider* son independientes de los interfaces genéricos definidos en el API y, en principio, no son accesibles directamente, son necesarios "pasos intermedios" para almacenar y recuperar las representaciones internas de las claves (especialmente en el caso de los pares de claves asimétricas).

- Todos los objetos que encapsulan algún tipo de clave (`SecretKey`, `PublicKey`, `PrivateKey`) disponen de un método `getEncoded()` que devuelve un `byte[]` con una representación independiente e interoperable de la propia clave (la *especificación* [*Spec*] de la clave).
- Uso de objetos `KeyFactory` y `SecretKeyFactory` para las creación de claves (`SecretKey`, `PublicKey`, `PrivateKey`) a partir de sus *especificaciones*
- Uso de `X509EncodedKeySpec` para encapsular la *especificación* de una clave pública (independiente del tipo de algoritmo "real") que se almacenará según el formato para certificados digitales X509
- Uso de `PKCS8EncodedKeySpec` para encapsular la *especificación* de una clave privada (independiente del tipo de algoritmo "real") que se almacenará según el formato para contenidos criptográficos PKCS8
- En las claves secretas se usa directamente la propia representación binaria de la clave.

Ejemplo de escritura/lectura de claves: AlmacenarClaves

```
$ javac -classpath ".:bcprov-jdk15on-xxx.jar" AlmacenarClaves.java (en Linux)
$ java -classpath ".:bcprov-jdk15on-xxx.jar" AlmacenarClaves (en Linux)
```

### 3.5. Firmas digitales

JCA ofrece objetos `Signature` para simplificar la generación y validación de firmas digitales

**Nota:** Otra alternativa es combinar "a mano" el uso de funciones HASH con objetos `Cipher`

- usando el modo cifrado con la clave privada para generar la firma digital a partir del HASH de los datos
- usando el modo descifrado con la clave pública para extraer el HASH firmado para posteriormente compararlo con el HASH calculado para validar la firma

**Creación de instancias** Empleando un método *factoria* (no con `new()`), ya que será el *provider* quien cree el objeto concreto con sus implementaciones de los correspondientes algoritmos.

Se debe indicar el nombre (alias) de la combinación de algoritmo de HASH y del algoritmo asimétrico a utilizar (MD5withRSA, MD5withDSA, SHA1withRSA, SHA1withDSA), etc.

Dependiendo del caso, suele ser necesario indicar también el nombre del *provider*.

```
static Signature getInstance(String algorithm)
static Signature getInstance(String algorithm, String provider)
```

### 3.5.1. Generación de firmas

**Configurar el modo "firma" del objeto *Signature*.** Mediante un método `initSign` al que se le indica la **clave privada** de firma.

```
void initSign(PrivateKey privateKey)
void initSign(PrivateKey privateKey, SecureRandom random)
```

**Alimentar los datos a firmar.** Pasando un array de `byte` mediante los métodos `update`.

```
void update(byte[] data)
void update(byte[] data, int off, int len)
```

**Generar la firma.** Generar y recuperar la firma con los métodos `sign()`.

```
byte[] sign()
int sign(byte[] outbuf, int offset, int len)
```

### 3.5.2. Verificación de firmas

**Configurar el modo "verificación" del objeto *Signature*.** Mediante un método `initVerify` al que se le indica la **clave pública** para validar la firma.

```
void initVerify(PublicKey publicKey)
void initVerify(Certificate certificate)
```

**Alimentar los datos a validar.** Pasando un array de `byte` mediante los métodos `update`.

```
void update(byte[] data)
void update(byte[] data, int off, int len)
```

**Verificar la firma.** Comprobar la validez de la firma con los métodos `verify()` pasando como argumento un array de `byte` con la firma a validar.

Devuelve un valor booleano indicado el resultado de la verificación (coincidencia o no de los HASHES).

```
boolean verify(byte[] signature)
boolean verify(byte[] signature, int offset, int length)
```