JCE stands for Java Cryptography Extension

**Java Cryptography Architecture**

The Java Cryptography Architecture (JCA) is the name for the internal design of the Java cryptography API.

JCA is structured around some central general purpose classes and interfaces. The real functionality behind these interfaces are provided by *providers*. Thus, you may use a Cipher class to encrypt and decrypt some data, but the concrete cipher implementation (encryption algorithm) depends on the concrete provider used.

You can implement and plugin your own providers too, but you should be careful with that. Implementing encryption correctly without security holes is hard! Unless you know what you are doing, you are probably better off using the builtin Java provider, or use a well established provider like Bouncy Castle.

## Core Classes and Interfaces

The Java cryptography API is divided between the following Java packages:

* java.security
* java.security.cert
* java.security.spec
* java.security.interfaces
* javax.crypto
* javax.crypto.spec
* javax.crypto.interfaces

The core classes and interfaces of these packages are:

* Provider
* SecureRandom
* Cipher
* MessageDigest
* Signature
* Mac
* AlgorithmParameters
* AlgorithmParameterGenerator
* KeyFactory
* SecretKeyFactory
* KeyPairGenerator
* KeyGenerator
* KeyAgreement
* KeyStore
* CertificateFactory
* CertPathBuilder
* CertPathValidator
* CertStore

## Provider

The Provider (java.security.Provider) class is a central class in the Java cryptography API. In order to use the Java crypto API you need a Provider set. The Java SDK comes with its own cryptography provider. If you don't set an explicit cryptography provider, the Java SDK default provider is used. However, this provider may not support the encryption algorithms you want to use. Therefore you might have to set your own cryptography provider.

One of the most popular cryptography providers for the Java cryptography API is called Bouncy Castle. Here is an example that sets a BouncyCastleProvider:

import org.bouncycastle.jce.provider.BouncyCastleProvider;

import java.security.Security;

public class ProviderExample {

public static void main(String[] args) {

Security.addProvider(new BouncyCastleProvider());

}

}

Understanding : The provider class is used to add the cryptography provider (Implementor). By default it uses Java Provider. The most commonly used provider *is Bouncy Castle Provider.*

## Cipher

The Cipher (javax.crypto.Cipher) class represents a cryptographic algorithm. A cipher can be used to both encrypt and decrypt data. The Cipher class is explained in more detail in the text on the [**Java Cipher**](http://tutorials.jenkov.com/java-cryptography/cipher.html)class, but I will give a brief introduction to the Cipher class in the following sections.

Here is how to create a Java Cipher instance:

Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

This example creates a Cipher instance which uses the AES encryption algorithm internally.

The Cipher.getInstance(...) method take a String identifying which encryption algorithm to use, as well as a few other configurations of the algorithm. In the example above, the CBC part is a mode the AES algorithm can work in. The PKCS5Padding part is how the AES algorithm should handle the last bytes of the data to encrypt, if the data does not align with a 64 bit or 128 bit block size boundary. What exactly that means belongs in a tutorial about cryptography in general, not a tutorial about the Java cryptography API.

Understanding : Cipher Class is used to create instance to encrypt and decrypt the data. It takes arguments which specifies the encryption algortithm used.

### Initializing the Cipher

Before the Cipher instance can be used it must be initialized. You initialize the Cipher instance by calling its init() method. The init() method takes two parameters:

* Encryption / Decryption cipher mode
* Key

The first parameter specifies whether the Cipher instance should encrypt or decrypt data. The second parameter specifies they key to use to encrypt or decrypt data with.

Here is a Java Cipher.init() example:

byte[] keyBytes = new byte[]{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

String algorithm = "RawBytes";

SecretKeySpec key = new SecretKeySpec(keyBytes, algorithm);

cipher.init(Cipher.ENCRYPT\_MODE, key);

Please note that the way the key is created in this example is not secure, and should not be used in practice. This Java cryptography tutorial will describe how to create keys more securely in sections later.

To initialize a Cipher instance to decrypt data you have to use the Cipher.DECRYPT\_MODE, like this:

cipher.init(Cipher.DECRYPT\_MODE, key);

### Encrypting or Decrypting Data

Once the Cipher is properly initialized you can start encrypting or decrypting data. You do so by calling the Cipher update() or doFinal() methods.

The update() method is used if you are encrypting or decrypting part of a bigger chunk of data. The doFinal() method is called when you are encrypting the last part of the big chunk of data, or if the block you pass to doFinal() represents the complete data block to encrypt.

Here is an example of encrypting some data with the doFinal() method

byte[] plainText = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");

byte[] cipherText = cipher.doFinal(plainText);

To decrypt data you would have passed cipher text (encrypted data) into the doFinal() or doUpdate()method instead.

## Keys

To encrypt or decrypt data you need a key. There are two types of keys - depending on which type of encryption algorithm you use:

* Symmetric keys
* Asymmetric keys

Symmetric keys are used for symmetric encryption algorithms. Symmetric encryption algorithms use the same key for encryption and decryption.

Asymmetric keys are used for asymmetric encryption algorithms. Asymmetric encryption algorithms use one key for encryption, and another for decryption. The public key - private key encryption algorithms are examples of asymmetric encryption algorithms.

Somehow the party that needs to decrypt data needs to know the key needed to decrypt the data. If the party decrypting the data is not the same as the party encrypting it, somehow these two parties need to agree on a key, or exchange the key. This is referred to as key exchange.

### Key Security

Keys should be hard to guess so an attacker cannot easily guess the encryption key. The example in the previous section about the Cipher class used a very simple, hardcoded key. This is not a good idea in practice. If they key is easy to guess, it is easy for an attacker to decrypt the encrypted data and possibly create fake messages herself.

It is important to make a key hard to guess. Thus, a key should consist of random bytes. The more random, the better, and the more bytes, the harder to guess because there are more possible combinations.

### Generating a Key

You can use the Java KeyGenerator class to generate more random encryption keys. The KeyGenerator is covered in a bit more detail in the text about the [**Java KeyGenerator**](http://tutorials.jenkov.com/java-cryptography/keygenerator.html), but I will show you an example of how to use it here.

Here is a Java KeyGenerator example:

KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");

SecureRandom secureRandom = new SecureRandom();

int keyBitSize = 256;

keyGenerator.init(keyBitSize, secureRandom);

SecretKey secretKey = keyGenerator.generateKey();

The resulting SecretKey instance can be passed to the Cipher.init() method, like this:

cipher.init(Cipher.ENCRYPT\_MODE, secretKey);

### Generating a Key Pair

Asymmetric encryption algorithms use a key pair consisting of a public key and a private key to encrypt and decrypt data. To generate an asymmetric key pair you can use the KeyPairGenerator(java.security.KeyPairGenerator). The KeyPairGenerator is covered in a bit more detail in [**Java KeyPairGenerator tutorial**](http://tutorials.jenkov.com/java-cryptography/keypairgenerator.html), but here is a simple Java KeyPairGenerator example:

SecureRandom secureRandom = new SecureRandom();

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("DSA");

KeyPair keyPair = keyPairGenerator.generateKeyPair();

## KeyStore

The Java KeyStore is a database that can contain keys. A Java KeyStore is represented by the KeyStore(java.security.KeyStore) class. A KeyStore can hold the following types of keys:

* Private keys
* Public keys + certificates
* Secret keys

Private and public keys are used in asymmetric encryption. A public key can have an associated certificate. A certificate is a document that verifies the identity of the person, organization or device claiming to own the public key. A certificate is typically digitally signed by the verifying party as proof.

Secret keys are used in symmetric encryption.

The KeyStore class is quite advanced so it is described in more detail in its own [**Java KeyStore Tutorial**](http://tutorials.jenkov.com/java-cryptography/keystore.html).

## Keytool

The Java Keytool is a command line tool that can work with Java KeyStore files. The Keytool can generate key pairs into a KeyStore file, export certificates from, and import certificates into a KeyStore and several other functions.

The Keytool comes with the Java installation. The Keytool is described in more detail in the tutorial about the [**Java Keytool**](http://tutorials.jenkov.com/java-cryptography/keytool.html).

## MessageDigest

When you receive some encrypted data from someone else, how do you know that no one has modified the encrypted data on the way to you?

A common solution is to calculate a message digest from the data before it is encrypted, and then encrypt both the data and the message digest and send that across the wire. A message digest is a hash value calculated from the message data. If a byte is changed in the encrypted data, the message digest calculated from the data will change too.

When receiving encrypted data, you decrypt it and calculate the message digest from it, and compare the calculated message digest to the message digest that was sent along with the encrypted data. If the two message digests are the same there is a high probability (but not a 100% guarantee) that the data was not modified.

You can use the Java MessageDigest (java.security.MessageDigest) to calculate message digests. You call the MessageDigest.getInstance() method to create a MessageDigest instance. There are several different message digest algorithms available. You need to tell which algorithm you want to use when creating a MessageDigest instance. This is covered in more detail in the [**Java MessageDigest**](http://tutorials.jenkov.com/java-cryptography/messagedigest.html) tutorial. Here is a short introduction to the MessageDigest class.

Here is an example of creating a MessageDigest example:

MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");

This example creates MessageDigest instance which uses the SHA-256 cryptographic hash algorithm internally to calculate message digests.

In order to calculate a message digest of some data you call the update() or digest() method.

The update() method can be called multiple times, and the message digest is updated internally. When you have passed all the data you want to include in the message digest, you call digest() and get the resulting message digest data out. Here is an example of calling update() several times followed by a digest() call:

MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");

byte[] data1 = "0123456789".getBytes("UTF-8");

byte[] data2 = "abcdefghijklmnopqrstuvxyz".getBytes("UTF-8");

messageDigest.update(data1);

messageDigest.update(data2);

byte[] digest = messageDigest.digest();

You can also call digest() a single time passing all the data to calculate the message digest from. Here is how that looks:

MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");

byte[] data1 = "0123456789".getBytes("UTF-8");

byte[] digest = messageDigest.digest(data1);

## Mac

The Java Mac class is used to create a MAC from a message. The term MAC is short for Message Authentication Code. A MAC is similar to a message digest, but uses an additional key to encrypt the message digest. Only by having both the original data and the key can you verify the MAC. Thus, a MAC is a more secure way to guard a block of data from modification than a message digest. The Mac class is described in more detail in the [**Java Mac**](http://tutorials.jenkov.com/java-cryptography/mac.html) tutorial, but below is a short introduction.

You create a Java Mac instance by calling the Mac.getInstance() method, passing as parameter the name of the algorithm to use. Here is how that looks:

Mac mac = Mac.getInstance("HmacSHA256");

Before you can create a MAC from data you must initialize the Mac instance with a key. Here is an example of initializing the Mac instance with a key:

byte[] keyBytes = new byte[]{0,1,2,3,4,5,6,7,8 ,9,10,11,12,13,14,15};

String algorithm = "RawBytes";

SecretKeySpec key = new SecretKeySpec(keyBytes, algorithm);

mac.init(key);

Once the Mac instance is initialized you can calculate a MAC from data by calling the update() and doFinal() method. If you have all the data to calculate the MAC for, you can call the doFinal() method immediately. Here is how that looks:

byte[] data = "abcdefghijklmnopqrstuvxyz".getBytes("UTF-8");

byte[] macBytes = mac.doFinal(data);

If you only have the access to the data in separate blocks, call update() multiple times with the data, and finish off with a call to doFinal(). Here is how that looks:

byte[] data = "abcdefghijklmnopqrstuvxyz".getBytes("UTF-8");

byte[] data2 = "0123456789".getBytes("UTF-8");

mac.update(data);

mac.update(data2);

byte[] macBytes = mac.doFinal();

## Signature

The Signature (java.security.Signature) class is used to digital sign data. When data is signed a digital signature is created from that data. The signature is thus separate from the data.

A digital signature is created by creating a message digest (hash) from the data, and encrypting that message digest with the private key of the device, person or organization that is to sign the data. The encrypted message digest is called a digital signature.

To create a Signature instance you call the Signature.getInstance(...) method. Here is an example that creates a Signature instance:

Signature signature = Signature.getInstance("SHA256WithDSA");

### Signing Data

To sign data you must initialize the Signature instance in signature mode. You do so by calling the initSign(...) method passing the private key to use to sign the data. Here is how initializing a Signatureinstance in signature mode is done:

signature.initSign(keyPair.getPrivate(), secureRandom);

Once the Signature instance is initialized it can be used to sign data. You do so by calling update()passing the data to sign as parameter. You can call the update() method several times with more data to include when creating the signature. When all the data has been passed to the update() method you call the sign() method to obtain the digital signature. Here is how that looks:

byte[] data = "abcdefghijklmnopqrstuvxyz".getBytes("UTF-8");

signature.update(data);

byte[] digitalSignature = signature.sign();

### Verifying a Signature

To verify a signature you must initialize a Signature instance into verification mode. This is done by calling the initVerify(...) method passing as parameter the public key to use to verify the signature. Here is now initializing a Signature instance into verification mode looks:

Signature signature = Signature.getInstance("SHA256WithDSA");

signature.initVerify(keyPair.getPublic());

Once initialized into verification mode, you call the update() method with the data the signature is signing, and finish with a call to verify() which returns true or false depending on whether the signature could be verified or not. Here is how verifying a signature looks:

byte[] data2 = "abcdefghijklmnopqrstuvxyz".getBytes("UTF-8");

signature2.update(data2);

boolean verified = signature2.verify(digitalSignature);

### Full Signature and Verification Example

Here is a full example of both creating and verifying a digital signature with the Signature class:

SecureRandom secureRandom = new SecureRandom();

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("DSA");

KeyPair keyPair = keyPairGenerator.generateKeyPair();

Signature signature = Signature.getInstance("SHA256WithDSA");

signature.initSign(keyPair.getPrivate(), secureRandom);

byte[] data = "abcdefghijklmnopqrstuvxyz".getBytes("UTF-8");

signature.update(data);

byte[] digitalSignature = signature.sign();

Signature signature2 = Signature.getInstance("SHA256WithDSA");

signature2.initVerify(keyPair.getPublic());

signature2.update(data);

boolean verified = signature2.verify(digitalSignature);

System.out.println("verified = " + verified);

Summary

Java Cipher – Class used for encryption and Decryption