# **Java Cipher**

The *Java Cipher* (javax.crypto.Cipher) class represents an encryption algorithm. The term *Cipher* is standard term for an encryption algorithm in the world of cryptography. That is why the Java class is called Cipher and not e.g. Encrypter / Decrypter or something else.

You can use a Cipher instance to encrypt and decrypt data in Java. This Java Cipher tutorial will explain how the Cipher class of the Java Cryptography API works.

**Creating a Cipher**

Before you can use a Java Cipher you just create an instance of the Cipher class. You create a Cipherinstance by calling its getInstance() method with a parameter telling what type of encryption algorithm you want to use. Here is an example of creating a Java Cipher instance:

Cipher cipher = Cipher.getInstance("AES");

This example creates a Cipher instance using the encryption algorithm called AES.

**Cipher Modes**

Some encryption algorithms can work in different modes. An encryption mode specifies details about how the algorithm should encrypt data. Thus, the encryption mode impacts part of the encryption algorithm.

The encryption modes can sometimes be used with multiple different encryption algorithms - like a technique that is appended to the core encryption algorithm. That is why the modes are thought of as separate from the encryption algorithms themselves, and rather "add-ons" to the encryption algorithms. Here are some of the most wellknown cipher modes:

* EBC - Electronic Codebook
* CBC - Cipher Block Chaining
* CFB - Cipher Feedback
* OFB - Output Feedback
* CTR - Counter

When instantiating a cipher you can append its mode to the name of the encryption algorithm. For instance, to create an AES Cipher instance using Cipher Block Chaining (CBC) you use this code:

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

Since Cipher Block Chaining requires a "padding scheme" too, the padding scheme is appended in the end of the encryption algorithm name string.

Please keep in mind that not all encryption algorithms and modes are supported by the default Java SDK cryptography provider. You might need an external provider like Bouncy Castle installed to create your desired Cipher instance with the required mode and padding scheme.

**Initializing a Cipher**

Before you can use a Cipher instance you must initialize it. Initializing a Cipher is done by calling its init() method. The init() method takes two parameters:

* Encryption / decryption cipher operation mode.
* Encryption / decryption key.

Here is an example of initializing a Cipher instance in encryption mode:

Key key = ... // get / create symmetric encryption key

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

Here is an example of initializing a Cipher instance in decryption mode:

Key key = ... // get / create symmetric encryption key

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

**Encrypting and Decrypting Data**

In order encrypt or decrypt data with a Cipher instance you call one of these two methods:

* update()
* doFinal()

There are several overridden versions of both update() and doFinal() which takes different parameters. I will cover the most commonly used versions here.

If you have to encrypt or decrypt a single block of data, just call the doFinal() with the data to encrypt or decrypt. Here is an encryption example:

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

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

The code actually looks pretty much the same in case of decrypting data. Just keep in mind that theCipher instance must be initialized into decryption mode. Here is how decrypting a single block of cipher text looks:

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

If you have to encrypt or decrypt multiple blocks of data, e.g. multiple blocks from a large file, you call the update() once for each block of data, and finish with a call to doFinal() with the last data block. Here is an example of encrypting multiple blocks of data:

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

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

byte[] data3 = "01234567890123456789012345".getBytes("UTF-8");

byte[] cipherText1 = cipher.update(data1);

byte[] cipherText2 = cipher.update(data2);

byte[] cipherText3 = cipher.doFinal(data3);

The reason a call to doFinal() is needed for the last block of data is, that some encryption algorithms need to pad the the data to fit a certain cipher block size (e.g. an 8 byte boundary). But - we do not want to pad the intermediate blocks of data encrypted. Hence the calls to update() for intermediate blocks of data, and the call to doFinal() for the last block of data.

When decrypting multiple blocks of data you also call the Cipher update() method for intermediate data blocks, and the doFinal() method for the last block. Here is an example of decrypting multiple blocks of data with a Java Cipher instance:

byte[] plainText1 = cipher.update(cipherText1);

byte[] plainText2 = cipher.update(cipherText2);

byte[] plainText3 = cipher.doFinal(cipherText3);

Again, the Cipher instance must be initialized into decryption mode for this example to work.

**Encrypting / Decrypting Part of a Byte Array**

The Java Cipher class encryption and decryption methods can encrypt or decrypt part of the data stored in a byte array. You simply pass an offset and length to the update() and / or doFinal() method. Here is an example:

int offset = 10;

int length = 24;

byte[] cipherText = cipher.doFinal(data, offset, length);

This example will encrypt (or decrypt, depending on the initialization of the Cipher) from byte with index 8 and 24 bytes forward.

**Encrypting / Decrypting Into an Existing Byte Array**

All the encryption and decryption examples shown in this tutorial so far have been returning the encrypted or decrypted data in a new byte array. However, it is also possible to encrypt or decrypt data into an existing byte array. This can be useful to keep the number of created byte arrays down.

You can encrypt or decrypt data into an existing byte array by passing the destination byte array as parameter to the update() and / or doFinal() method. Here is an example:

int offset = 10;

int length = 24;

byte[] dest = new byte[1024];

cipher.doFinal(data, offset, length, dest);

This example encrypts the data from the byte with index 10 and 24 bytes forward into the dest byte array from offset 0. If you want to set a different offset for the dest byte array there is a version of update() and doFinal() which takes an offset parameter extra. Here is an example of calling the doFinal() method with an offset into the dest array:

int offset = 10;

int length = 24;

byte[] dest = new byte[1024];

int destOffset = 12

cipher.doFinal(data, offset, length, dest, destOffset);

**Reusing a Cipher Instance**

Initializing a Java Cipher instance is an expensive operation. Therefore it is a good idea to reuse Cipherinstances. Luckily, the Cipher class was designed with reuse in mind.

When you call the doFinal() method on a Cipher instance, the Cipher instance is returned to the state it had just after initialization. The Cipher instance can then be used to encrypt or decrypt more data again.

Here is an example of reusing a Java Cipher instance:

Cipher cipher = Cipher.getInstance("AES");

Key key = ... // get / create symmetric encryption key

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

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

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

byte[] cipherText1 = cipher.update(data1);

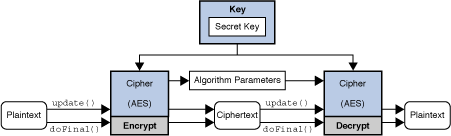
byte[] cipherText2 = cipher.doFinal(data2);

byte[] data3 = "01234567890123456789012345".getBytes("UTF-8");

byte[] cipherText3 = cipher.doFinal(data3);

First the Cipher instance is created and initialized, and then used to encrypt two blocks of coherent data. Notice the call to update() and then doFinal() for these two blocks of data. Now the Cipher instance can be used again to encrypt more data. This is done with the doFinal() call with the third data block. After this doFinal() call you can encrypt yet another block of data with the same Java Cipher instance.

The Cipher class provides the functionality of a cryptographic cipher used for encryption and decryption. Encryption is the process of taking data (called *cleartext*) and a *key*, and producing data (*ciphertext*) meaningless to a third-party who does not know the key. Decryption is the inverse process: that of taking ciphertext and a key and producing cleartext.



[Description of Figure 7: The Cipher Class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "cipher)

### Symmetric vs. Asymmetric Cryptography

There are two major types of encryption: *symmetric* (also known as *secret key*), and *asymmetric* (or *public key cryptography*). In symmetric cryptography, the same secret key to both encrypt and decrypt the data. Keeping the key private is critical to keeping the data confidential. On the other hand, asymmetric cryptography uses a public/private key pair to encrypt data. Data encrypted with one key is decrypted with the other. A user first generates a public/private key pair, and then publishes the public key in a trusted database that anyone can access. A user who wishes to communicate securely with that user encrypts the data using the retrieved public key. Only the holder of the private key will be able to decrypt. Keeping the private key confidential is critical to this scheme.

Asymmetric algorithms (such as RSA) are generally much slower than symmetric ones. These algorithms are not designed for efficiently protecting large amounts of data. In practice, asymmetric algorithms are used to exchange smaller secret keys which are used to initialize symmetric algorithms.

### Stream vs. Block Ciphers

There are two major types of ciphers: *block* and *stream*. Block ciphers process entire blocks at a time, usually many bytes in length. If there is not enough data to make a complete input block, the data must be *padded*: that is, before encryption, dummy bytes must be added to make a multiple of the cipher's block size. These bytes are then stripped off during the decryption phase. The padding can either be done by the application, or by initializing a cipher to use a padding type such as "PKCS5PADDING". In contrast, stream ciphers process incoming data one small unit (typically a byte or even a bit) at a time. This allows for ciphers to process an arbitrary amount of data without padding.

### Modes Of Operation

When encrypting using a simple block cipher, two identical blocks of plaintext will always produce an identical block of cipher text. Cryptanalysts trying to break the ciphertext will have an easier job if they note blocks of repeating text. In order to add more complexity to the text, feedback modes use the previous block of output to alter the input blocks before applying the encryption algorithm. The first block will need an initial value, and this value is called the *initialization vector (IV)*. Since the IV simply alters the data before any encryption, the IV should be random but does not necessarily need to be kept secret. There are a variety of modes, such as CBC (Cipher Block Chaining), CFB (Cipher Feedback Mode), and OFB (Output Feedback Mode). ECB (Electronic Codebook Mode) is a mode in which there is no influence from block position or other ciphertext blocks. Because ECB ciphertexts are the same if they use the same plaintext/key, this mode is not typically suitable for cryptographic applications and should not be used.

Some algorithms such as AES and RSA allow for keys of different lengths, but others are fixed, such as 3DES. Encryption using a longer key generally implies a stronger resistance to message recovery. As usual, there is a trade off between security and time, so choose the key length appropriately.

Most algorithms use binary keys. Most humans do not have the ability to remember long sequences of binary numbers, even when represented in hexadecimal. Character passwords are much easier to recall. Because character passwords are generally chosen from a small number of characters (for example, [a-zA-Z0-9]), protocols such as "Password-Based Encryption" (PBE) have been defined which take character passwords and generate strong binary keys. In order to make the task of getting from password to key very time-consuming for an attacker (via so-called "dictionary attacks" where common dictionary word->value mappings are precomputed), most PBE implementations will mix in a random number, known as a *salt*, to increase the key randomness.

Newer cipher modes such as Authenticated Encryption with Associated Data (AEAD) (for example, [Galois/Counter Mode (GCM)](http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf)) encrypt data and authenticate the resulting message simultaneously. Additional Associated Data (AAD) can be used during the calculation of the resulting AEAD tag (Mac), but this AAD data is not output as ciphertext. (For example, some data might not need to be kept confidential, but should figure into the tag calculation to detect modifications.) The Cipher.updateAAD() methods can be used to include AAD in the tag calculations.

#### Creating a Cipher Object

Cipher objects are obtained by using one of the Cipher [getInstance() static factory methods](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderImplReq). Here, the algorithm name is slightly different than with other engine classes, in that it specifies not just an algorithm name, but a "transformation". A transformation is a string that describes the operation (or set of operations) to be performed on the given input to produce some output. A transformation always includes the name of a cryptographic algorithm (e.g., AES), and may be followed by a mode and padding scheme.

A transformation is of the form:

* "*algorithm/mode/padding*" or
* "*algorithm*"

For example, the following are valid transformations:

"*AES/CBC/PKCS5Padding*"

"*AES*"

If just a transformation name is specified, the system will determine if there is an implementation of the requested transformation available in the environment, and if there is more than one, returns there is a preferred one.

If both a transformation name and a package provider are specified, the system will determine if there is an implementation of the requested transformation in the package requested, and throw an exception if there is not.

It is recommended to use a transformation that fully specifies the algorithm, mode, and padding. By not doing so, the provider will use a default. For example, the SunJCE and SunPKCS11 providers uses ECB as the default mode, and PKCS5Padding as the default padding for many symmetric ciphers.

This means that in the case of the SunJCE provider:

Cipher c1 = Cipher.getInstance("*AES/ECB/PKCS5Padding*");

and

Cipher c1 = Cipher.getInstance("*AES*");

are equivalent statements.

**Note:** ECB mode is the easiest block cipher mode to use and is the default in the JDK/JRE. ECB works well for single blocks of data, but absolutely should not be used for multiple data blocks.

Using modes such as CFB and OFB, block ciphers can encrypt data in units smaller than the cipher's actual block size. When requesting such a mode, you may optionally specify the number of bits to be processed at a time by appending this number to the mode name as shown in the "*AES/CFB8/NoPadding*" and "*AES/OFB32/PKCS5Padding*" transformations. If no such number is specified, a provider-specific default is used. (For example, the SunJCE provider uses a default of 128 bits for AES.) Thus, block ciphers can be turned into byte-oriented stream ciphers by using an 8 bit mode such as CFB8 or OFB8.

[Appendix A](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#AppA) of this document contains a list of standard names that can be used to specify the algorithm name, mode, and padding scheme components of a transformation.

The objects returned by factory methods are uninitialized, and must be initialized before they become usable.

#### Initializing a Cipher Object

A Cipher object obtained via getInstance must be initialized for one of four modes, which are defined as final integer constants in the Cipher class. The modes can be referenced by their symbolic names, which are shown below along with a description of the purpose of each mode:

**ENCRYPT\_MODE**

Encryption of data.

**DECRYPT\_MODE**

Decryption of data.

**WRAP\_MODE**

Wrapping a java.security.Key into bytes so that the key can be securely transported.

**UNWRAP\_MODE**

Unwrapping of a previously wrapped key into a java.security.Key object.

Each of the Cipher initialization methods takes an operational mode parameter (opmode), and initializes the Cipher object for that mode. Other parameters include the key (key) or certificate containing the key (certificate), algorithm parameters (params), and a source of randomness (random).

To initialize a Cipher object, call one of the following init methods:

public void init(int opmode, Key key);

public void init(int opmode, Certificate certificate);

public void init(int opmode, Key key, SecureRandom random);

public void init(int opmode, Certificate certificate,

SecureRandom random);

public void init(int opmode, Key key,

AlgorithmParameterSpec params);

public void init(int opmode, Key key,

AlgorithmParameterSpec params, SecureRandom random);

public void init(int opmode, Key key,

AlgorithmParameters params);

public void init(int opmode, Key key,

AlgorithmParameters params, SecureRandom random);

If a Cipher object that requires parameters (e.g., an initialization vector) is initialized for encryption, and no parameters are supplied to the init method, the underlying cipher implementation is supposed to supply the required parameters itself, either by generating random parameters or by using a default, provider-specific set of parameters.

However, if a Cipher object that requires parameters is initialized for decryption, and no parameters are supplied to the init method, an InvalidKeyException or InvalidAlgorithmParameterException exception will be raised, depending on the init method that has been used.

See the section about [Managing Algorithm Parameters](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ManagingParameters) for more details.

The same parameters that were used for encryption must be used for decryption.

Note that when a Cipher object is initialized, it loses all previously-acquired state. In other words, initializing a Cipher is equivalent to creating a new instance of that Cipher, and initializing it. For example, if a Cipher is first initialized for decryption with a given key, and then initialized for encryption, it will lose any state acquired while in decryption mode.

#### Encrypting and Decrypting Data

Data can be encrypted or decrypted in one step (*single-part operation*) or in multiple steps (*multiple-part operation*). A multiple-part operation is useful if you do not know in advance how long the data is going to be, or if the data is too long to be stored in memory all at once.

To encrypt or decrypt data in a single step, call one of the doFinal methods:

public byte[] doFinal(byte[] input);

public byte[] doFinal(byte[] input, int inputOffset, int inputLen);

public int doFinal(byte[] input, int inputOffset,

int inputLen, byte[] output);

public int doFinal(byte[] input, int inputOffset,

int inputLen, byte[] output, int outputOffset)

To encrypt or decrypt data in multiple steps, call one of the update methods:

public byte[] update(byte[] input);

public byte[] update(byte[] input, int inputOffset, int inputLen);

public int update(byte[] input, int inputOffset, int inputLen,

byte[] output);

public int update(byte[] input, int inputOffset, int inputLen,

byte[] output, int outputOffset)

A multiple-part operation must be terminated by one of the above doFinal methods (if there is still some input data left for the last step), or by one of the following doFinal methods (if there is no input data left for the last step):

public byte[] doFinal();

public int doFinal(byte[] output, int outputOffset);

All the doFinal methods take care of any necessary padding (or unpadding), if padding (or unpadding) has been requested as part of the specified transformation.

A call to doFinal resets the Cipher object to the state it was in when initialized via a call to init. That is, the Cipher object is reset and available to encrypt or decrypt (depending on the operation mode that was specified in the call to init) more data.

#### Wrapping and Unwrapping Keys

Wrapping a key enables secure transfer of the key from one place to another.

The wrap/unwrap API makes it more convenient to write code since it works with key objects directly. These methods also enable the possibility of secure transfer of hardware-based keys.

To **wrap** a Key, first initialize the Cipher object for WRAP\_MODE, and then call the following:

public final byte[] wrap(Key key);

If you are supplying the wrapped key bytes (the result of calling wrap) to someone else who will unwrap them, be sure to also send additional information the recipient will need in order to do the unwrap:

1. the name of the key algorithm, and
2. the type of the wrapped key (one of Cipher.SECRET\_KEY, Cipher.PRIVATE\_KEY, or Cipher.PUBLIC\_KEY).

The key algorithm name can be determined by calling the getAlgorithm method from the Key interface:

public String getAlgorithm();

To **unwrap** the bytes returned by a previous call to wrap, first initialize a Cipher object for UNWRAP\_MODE, then call the following:

public final Key unwrap(byte[] wrappedKey,

String wrappedKeyAlgorithm,

int wrappedKeyType));

Here, wrappedKey is the bytes returned from the previous call to wrap, wrappedKeyAlgorithm is the algorithm associated with the wrapped key, and wrappedKeyType is the type of the wrapped key. This must be one of Cipher.SECRET\_KEY, Cipher.PRIVATE\_KEY, or Cipher.PUBLIC\_KEY.

#### Managing Algorithm Parameters

The parameters being used by the underlying Cipher implementation, which were either explicitly passed to the init method by the application or generated by the underlying implementation itself, can be retrieved from the Cipher object by calling its getParameters method, which returns the parameters as ajava.security.AlgorithmParameters object (or null if no parameters are being used). If the parameter is an initialization vector (IV), it can also be retrieved by calling the getIV method.

In the following example, a Cipher object implementing password-based encryption (PBE) is initialized with just a key and no parameters. However, the selected algorithm for password-based encryption requires two parameters - a *salt* and an *iteration count*. Those will be generated by the underlying algorithm implementation itself. The application can retrieve the generated parameters from the Cipher object as follows:

import javax.crypto.\*;

import java.security.AlgorithmParameters;

// get cipher object for password-based encryption

Cipher c = Cipher.getInstance("PBEWithMD5AndDES");

// initialize cipher for encryption, without supplying

// any parameters. Here, "myKey" is assumed to refer

// to an already-generated key.

c.init(Cipher.ENCRYPT\_MODE, myKey);

// encrypt some data and store away ciphertext

// for later decryption

byte[] cipherText = c.doFinal("This is just an example".getBytes());

// retrieve parameters generated by underlying cipher

// implementation

AlgorithmParameters algParams = c.getParameters();

// get parameter encoding and store it away

byte[] encodedAlgParams = algParams.getEncoded();

The same parameters that were used for encryption must be used for decryption. They can be instantiated from their encoding and used to initialize the corresponding Cipher object for decryption, as follows:

import javax.crypto.\*;

import java.security.AlgorithmParameters;

// get parameter object for password-based encryption

AlgorithmParameters algParams;

algParams = AlgorithmParameters.getInstance("PBEWithMD5AndDES");

// initialize with parameter encoding from above

algParams.init(encodedAlgParams);

// get cipher object for password-based encryption

Cipher c = Cipher.getInstance("PBEWithMD5AndDES");

// initialize cipher for decryption, using one of the

// init() methods that takes an AlgorithmParameters

// object, and pass it the algParams object from above

c.init(Cipher.DECRYPT\_MODE, myKey, algParams);

If you did not specify any parameters when you initialized a Cipher object, and you are not sure whether or not the underlying implementation uses any parameters, you can find out by simply calling the getParameters method of your Cipher object and checking the value returned. A return value of null indicates that no parameters were used.

The following cipher algorithms implemented by the SunJCE provider use parameters:

* AES, DES-EDE, and Blowfish, when used in feedback (i.e., CBC, CFB, OFB, or PCBC) mode, use an initialization vector (IV). The javax.crypto.spec.IvParameterSpec class can be used to initialize a Cipher object with a given IV.
* PBEWithMD5AndDES uses a set of parameters, comprising a salt and an iteration count. The javax.crypto.spec.PBEParameterSpec class can be used to initialize a Cipher object implementing PBEWithMD5AndDES with a given salt and iteration count.

Note that you do not have to worry about storing or transferring any algorithm parameters for use by the decryption operation if you use the [SealedObject](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#SealedObject) class. This class attaches the parameters used for sealing (encryption) to the encrypted object contents, and uses the same parameters for unsealing (decryption).

#### Cipher Output Considerations

Some of the update and doFinal methods of Cipher allow the caller to specify the output buffer into which to encrypt or decrypt the data. In these cases, it is important to pass a buffer that is large enough to hold the result of the encryption or decryption operation.

The following method in Cipher can be used to determine how big the output buffer should be:

public int getOutputSize(int inputLen)

## Other Cipher-based Classes

There are some helper classes which internally use Ciphers to provide easy access to common cipher uses.

### The Cipher Stream Classes

#### The CipherInputStream Class

This class is a FilterInputStream that encrypts or decrypts the data passing through it. It is composed of an InputStream, or one of its subclasses, and a Cipher. CipherInputStream represents a secure input stream into which a Cipher object has been interposed. The read methods of CipherInputStream return data that are read from the underlying InputStream but have additionally been processed by the embedded Cipher object. The Cipher object must be fully initialized before being used by a CipherInputStream.

For example, if the embedded Cipher has been initialized for decryption, the CipherInputStream will attempt to decrypt the data it reads from the underlying InputStream before returning them to the application.

This class adheres strictly to the semantics, especially the failure semantics, of its ancestor classes java.io.FilterInputStream and java.io.InputStream. This class has exactly those methods specified in its ancestor classes, and overrides them all, so that the data are additionally processed by the embedded cipher. Moreover, this class catches all exceptions that are not thrown by its ancestor classes. In particular, the skip(long) method skips only data that has been processed by the Cipher.

It is crucial for a programmer using this class not to use methods that are not defined or overridden in this class (such as a new method or constructor that is later added to one of the super classes), because the design and implementation of those methods are unlikely to have considered security impact with regard to CipherInputStream.

As an example of its usage, suppose cipher1 has been initialized for encryption. The code below demonstrates how to use a CipherInputStream containing that cipher and a FileInputStream in order to encrypt input stream data:

FileInputStream fis;

FileOutputStream fos;

CipherInputStream cis;

fis = new FileInputStream("/tmp/a.txt");

cis = new CipherInputStream(fis, cipher1);

fos = new FileOutputStream("/tmp/b.txt");

byte[] b = new byte[8];

int i = cis.read(b);

while (i != -1) {

fos.write(b, 0, i);

i = cis.read(b);

}

fos.close();

The above program reads and encrypts the content from the file /tmp/a.txt and then stores the result (the encrypted bytes) in /tmp/b.txt.

The following example demonstrates how to easily connect several instances of CipherInputStream and FileInputStream. In this example, assume that cipher1 and cipher2 have been initialized for encryption and decryption (with corresponding keys), respectively.

FileInputStream fis;

FileOutputStream fos;

CipherInputStream cis1, cis2;

fis = new FileInputStream("/tmp/a.txt");

cis1 = new CipherInputStream(fis, cipher1);

cis2 = new CipherInputStream(cis1, cipher2);

fos = new FileOutputStream("/tmp/b.txt");

byte[] b = new byte[8];

int i = cis2.read(b);

while (i != -1) {

fos.write(b, 0, i);

i = cis2.read(b);

}

fos.close();

The above program copies the content from file /tmp/a.txt to /tmp/b.txt, except that the content is first encrypted and then decrypted back when it is read from /tmp/a.txt. Of course since this program simply encrypts text and decrypts it back right away, it's actually not very useful except as a simple way of illustrating chaining of CipherInputStreams.

Note that the read methods of the CipherInputStream will block until data is returned from the underlying cipher. If a block cipher is used, a full block of cipher text will have to be obtained from the underlying InputStream.

#### The CipherOutputStream Class

This class is a FilterOutputStream that encrypts or decrypts the data passing through it. It is composed of an OutputStream, or one of its subclasses, and a Cipher. CipherOutputStream represents a secure output stream into which a Cipher object has been interposed. The write methods of CipherOutputStream first process the data with the embedded Cipher object before writing them out to the underlying OutputStream. The Cipher object must be fully initialized before being used by a CipherOutputStream.

For example, if the embedded Cipher has been initialized for encryption, the CipherOutputStream will encrypt its data, before writing them out to the underlying output stream.

This class adheres strictly to the semantics, especially the failure semantics, of its ancestor classes java.io.OutputStream and java.io.FilterOutputStream. This class has exactly those methods specified in its ancestor classes, and overrides them all, so that all data are additionally processed by the embedded cipher. Moreover, this class catches all exceptions that are not thrown by its ancestor classes.

It is crucial for a programmer using this class not to use methods that are not defined or overridden in this class (such as a new method or constructor that is later added to one of the super classes), because the design and implementation of those methods are unlikely to have considered security impact with regard to CipherOutputStream.

As an example of its usage, suppose cipher1 has been initialized for encryption. The code below demonstrates how to use a CipherOutputStream containing that cipher and a FileOutputStream in order to encrypt data to be written to an output stream:

FileInputStream fis;

FileOutputStream fos;

CipherOutputStream cos;

fis = new FileInputStream("/tmp/a.txt");

fos = new FileOutputStream("/tmp/b.txt");

cos = new CipherOutputStream(fos, cipher1);

byte[] b = new byte[8];

int i = fis.read(b);

while (i != -1) {

cos.write(b, 0, i);

i = fis.read(b);

}

cos.flush();

The above program reads the content from the file /tmp/a.txt, then encrypts and stores the result (the encrypted bytes) in /tmp/b.txt.

The following example demonstrates how to easily connect several instances of CipherOutputStream and FileOutputStream. In this example, assume that cipher1 and cipher2 have been initialized for decryption and encryption (with corresponding keys), respectively:

FileInputStream fis;

FileOutputStream fos;

CipherOutputStream cos1, cos2;

fis = new FileInputStream("/tmp/a.txt");

fos = new FileOutputStream("/tmp/b.txt");

cos1 = new CipherOutputStream(fos, cipher1);

cos2 = new CipherOutputStream(cos1, cipher2);

byte[] b = new byte[8];

int i = fis.read(b);

while (i != -1) {

cos2.write(b, 0, i);

i = fis.read(b);

}

cos2.flush();

The above program copies the content from file /tmp/a.txt to /tmp/b.txt, except that the content is first encrypted and then decrypted back before it is written to /tmp/b.txt.

One thing to keep in mind when using ***block*** cipher algorithms is that a full block of plaintext data must be given to the CipherOutputStream before the data will be encrypted and sent to the underlying output stream.

There is one other important difference between the flush and close methods of this class, which becomes even more relevant if the encapsulated Cipher object implements a block cipher algorithm with padding turned on:

* flush flushes the underlying OutputStream by forcing any buffered output bytes that have already been processed by the encapsulated Cipher object to be written out. Any bytes buffered by the encapsulated Cipher object and waiting to be processed by it will **not** be written out.
* close closes the underlying OutputStream and releases any system resources associated with it. It invokes the doFinal method of the encapsulated Cipher object, causing any bytes buffered by it to be processed and written out to the underlying stream by calling its flush method.