2

Symmetric Key Cryptography

Symmetric key ciphers are one of the workhorses of cryptography. They are used to secure bulk data, provide a foundation for message authentication codes, and provide support for password-based encryption as well. As symmetric key cryptography gains its security from keeping a shared key secret, it is also often referred to as *secret key* cryptography, a term that you will see is used in the JCE.

This chapter introduces the concept of symmetric key cryptography and how it is used in the JCE. I will cover creation of keys for symmetric key ciphers, creating Cipher objects to be used for encryption, how modes and padding mechanisms are specified in Java, what other parameter objects can be used to initialize ciphers and what they mean, how password-based encryption is used, methods for doing key wrapping, and how to do cipher-based I/O.

By the end of this chapter you should

- ❑ Be well equipped to make use of a variety of symmetric key ciphers
- ❑ Understand the various cipher modes and paddings and what they are for
- ❑ Be able to construct or randomly generate symmetric keys
- ❑ Understand key wrapping
- ❑ Be able to utilize the I/O classes provided in the JCE

Finally, you should also have a few ideas about where to look when you are trying to debug applications using symmetric key ciphers and what might go wrong with them.

A First Example

To get anywhere in this area, you have to first be able to create a key, and then you have to be able to create a cipher so that you can actually do something with it. If you recall the policy file test in the last chapter, you will remember it used two classes, `javax.crypto.Cipher` and `javax.crypto.spec.SecretKeySpec`. These two classes provide you with enough of a starting point to write a simple example program.

A Basic Utility Class

In the policy test program, you were mainly interested in whether you could create a cipher with a given key size and use it. This time you will carry out a simple encryption/decryption process so you can see how ciphers get used from end to end. Before you can do this, you need to define some basic infrastructure that allows you to look at the output of your programs easily. Encrypted data, as you can imagine, is only human-readable by chance, so for the purposes of investigation, it is best to print the bytes you are interested in using hex, which, being base-16, nicely maps to two digits a byte.

Here is a simple utility class for doing hex printing of a byte array:

```
package chapter2;

/**
 * General utilities for the second chapter examples.
 */
public class Utils
{
    private static String   digits = "0123456789abcdef";

    /**
     * Return length many bytes of the passed in byte array as a hex string.
     *
     * @param data the bytes to be converted.
     * @param length the number of bytes in the data block to be converted.
     * @return a hex representation of length bytes of data.
     */
    public static String toHex(byte[] data, int length)
    {
        StringBuffer    buf = new StringBuffer();

        for (int i = 0; i != length; i++)
        {
            int    v = data[i] & 0xff;

            buf.append(digits.charAt(v >> 4));
            buf.append(digits.charAt(v & 0xf));
        }

        return buf.toString();
    }

    /**
     * Return the passed in byte array as a hex string.
     *
     * @param data the bytes to be converted.
     * @return a hex representation of data.
     */
    public static String toHex(byte[] data)
    {
        return toHex(data, data.length);
    }
}
```

Copy and compile the utility class. Now you have done that, look at the example that follows.

Try It Out Using AES

Because this example is fairly simple, I'll explain the API I am using after it. However, you should note that the example is using an algorithm called AES. Prior to November 2001, the stock standard algorithm for doing symmetric key encryption was the Data Encryption Standard (DES) and a variant on it, namely, Triple-DES or DES-EDE. Now, following the announcement of the Advanced Encryption Standard (AES), your general preference should be to use AES. You will look at some other algorithms a bit later; however, a lot of work went into the development and selection of AES. It makes sense to take advantage of it, so AES is what you'll use in this example.

```
package chapter2;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;


/**
 * Basic symmetric encryption example
 */
public class SimpleSymmetricExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]        input = new byte[] {
                            0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
                            (byte)0x88, (byte)0x99, (byte)0xaa, (byte)0xbb,
                            (byte)0xcc, (byte)0xdd, (byte)0xee, (byte)0xff };
        byte[]        keyBytes = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                            0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };

        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");

        Cipher        cipher = Cipher.getInstance("AES/ECB/NoPadding", "BC");

        System.out.println("input text : " + Utils.toHex(input));

        // encryption pass

        byte[] cipherText = new byte[input.length];

        cipher.init(Cipher.ENCRYPT_MODE, key);

        int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

        ctLength += cipher.doFinal(cipherText, ctLength);

        System.out.println("cipher text: " + Utils.toHex(cipherText)
                                                    + " bytes: " + ctLength);
```

```
            // decryption pass

            byte[] plainText = new byte[ctLength];

            cipher.init(Cipher.DECRYPT_MODE, key);

            int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);

            ptLength += cipher.doFinal(plainText, ptLength);

            System.out.println("plain text : " + Utils.toHex(plainText)
                                                    + " bytes: " + ptLength);
        }
    }
```

Readers who also spend their time browsing through the NIST FIPS publications may recognize this as one of the standard vector tests for AES in FIPS-197. As an aside, if you are planning to get seriously involved in this area, you would do well to have some familiarity with the relevant NIST FIPS publications. The most relevant ones have been listed in Appendix D, and amongst other things, they are a big help if you need to confirm for yourself the validity of an implementation of an algorithm they describe.

If all is going well, and your class path is appropriately set up, when you run the program using

```
java chapter2.SimpleSymmetricExample
```

you will see

```
input text : 0112233445566778899aabbccddeeff
cipher text: dda97ca4864cdfe06eaf70a0ecd7191 bytes: 16
plain text : 0112233445566778899aabbccddeeff bytes: 16
```

You may also get the exception:

```
Exception in thread "main" java.security.NoSuchProviderException:
                                                Provider 'BC' not found
```

which means the provider is not properly installed.

Or you may get the exception:

```
Exception in thread "main" java.lang.SecurityException:
                            Unsupported keysize or algorithm parameters
```

which instead means it can find the provider, but the unrestricted policy files are not installed.

If you see either of these exceptions, or have any other problems, look through Chapter 1 again and make sure the Bouncy Castle provider has been correctly installed and the Java environment is correctly configured.

On the other hand, if everything is working, it is probably time you looked at how the program works.

How It Works

As you can see, the example demonstrates that carrying out a symmetric key encryption operation is a matter of providing a key to use and a suitable object for doing the processing on the input data, be it plaintext to be encrypted or ciphertext to be decrypted. In Java the easiest way to construct a symmetric key by hand is to use the `SecretKeySpec` class.

The SecretKeySpec Class

The `javax.crypto.spec.SecretKeySpec` class provides a simple mechanism for converting byte data into a secret key suitable for passing to a `Cipher` object's `init()` method. As you'll see a bit later, it is not the only way of creating a secret key, but it is certainly one that is used a lot. Looking at the previous Try It Out ("Using AES"), you can see that constructing a secret key can be as simple as passing a byte array and the name of the algorithm the key is to be used with. For more details on the class, see the JavaDoc that comes with the JCE.

> **One thing the `SecretKeySpec` will not do is stop you from passing a weak key to a `Cipher` object. Weak keys are keys that, for a given algorithm, do not provide strong cryptography and should be avoided. Not all algorithms have weak keys, but if you are using one that does, such as DES, you should take care to ensure that the bytes produced for creating the `SecretKeySpec` are not weak keys.**

The Cipher Class

A look at the previous example program quickly reveals that the creation and use of a `javax.crypto.Cipher` object follows a simple pattern. You create one using `Cipher.getInstance()`, initialize it with the mode you want using `Cipher.init()`, feed the input data in while collecting output at the same time using `Cipher.update()`, and then finish off the process with `Cipher.doFinal()`.

Cipher.getInstance()

A `Cipher` object, rather than being created using a constructor directly, is created using the static factory method `Cipher.getInstance()`. In the case of the example, it was done by passing two arguments, one giving the kind of cipher you want to create, the other giving the provider you want to use to create it — given by the name `"BC"`.

In the case of the cipher name `"AES/ECB/NoPadding"`, the name is composed of three parts. The first part is the name of algorithm — AES. The second part is the mode in which the algorithm should be used, ECB, or Electronic Code Book mode. Finally, the string `"NoPadding"` tells the provider you do not wish to use padding with this algorithm. Just ignore the mode and padding, as you will be reading about them soon. What is most important now is that when the *full name* of the `Cipher` object you want to be created is given, it always follows the *`AlgorithmName/Mode/TypeOfPadding`* pattern. You can also just give the algorithm name and provider, as in:

```
Cipher.getInstance("AES", "BC");
```

However if you do, it is purely up to the provider you are using as to which mode and padding will be used in the `Cipher` object that has been returned. It is advised against doing this in the interests of allowing your code to be portable between providers. Specify exactly what you need and you should be spared unnecessary surprises.

Cipher.init()

Having created a `Cipher` object using `Cipher.getInstance()` at a minimum, you then have to initialize it with the type of operation it is to be used for and with the key that is to be used. `Cipher` objects have four possible modes, all specified using static constants on the `Cipher` class. Two of the modes are connected with key wrapping, which you'll look at later, and the other two are `Cipher.ENCRYPT_MODE` and `Cipher.DECRYPT_MODE`, which were used previously. The `Cipher.init()` method can take other parameters as well, which you'll look at later. For the moment it is enough to understand that if you do not call the `init` method, any attempt to use the `Cipher` object will normally result in an `IllegalStateException`.

Cipher.update()

Once the `Cipher` object is set up for encryption or decryption, you can feed data into it and accept data from it. There are several convenience update methods on the `Cipher` class that you can read about in the JavaDoc, but the one used in the example is the most fundamental. Consider the line:

```
int ctLength = cipher.update(input, 0, input.length, cipherText, 0);
```

Cipher objects usually acquire a chunk of data, process it by copying the result into the output array (the argument `cipherText`), and then copy the next chunk and continue, filling the output array as they go. Thus, you cannot be sure how much data will be written each time you do an update; the number of output bytes may be O (zero), or it may be between 0 and the length of the input. The starting offset that the processed blocks are written to is the last argument to the method, in this case 0. Regardless of how many bytes get output during an update, you will only know how many bytes have been written to the output array if you keep track of the return value.

Cipher.doFinal()

Now consider the line:

```
ctLength += cipher.doFinal(cipherText, ctLength);
```

`Cipher.doFinal()` is very similar to `Cipher.update()` in that it may also put out 0 or more bytes, depending on the kind of `Cipher` object you specified with `Cipher.getInstance()`. Likewise, it also has a return value to tell you how many bytes it actually wrote to the output array (again the `cipherText` array). Note that the second argument is the offset at which writing of the output will start and is a value that has been preserved from the last `Cipher.update()`.

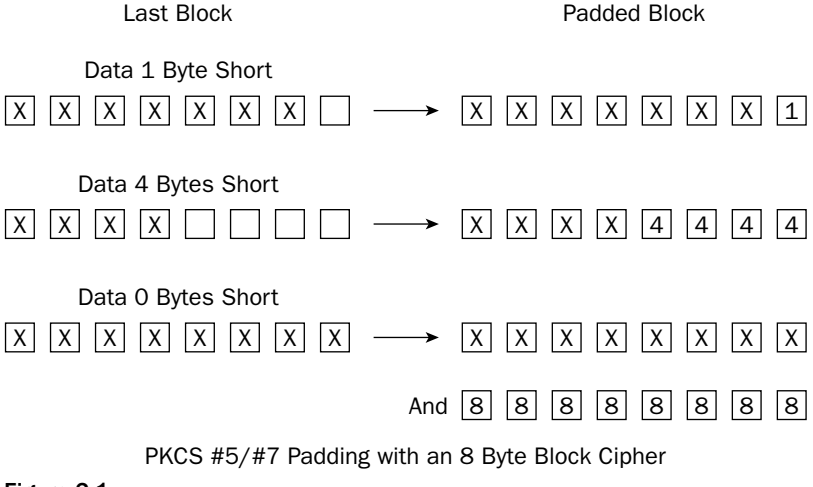
> **Failing to keep track of the return values from the `int` returning methods of `update()` and `doFinal()` is one of the most common error programmers make using the `Cipher` class. It is never okay to ignore the return values if you want to write flexible code.**

Symmetric Block Cipher Padding

While the value of the test vector used in the last example was not due to random chance, neither was the length of it. Most of the popular ciphers are block ciphers, and their block size is normally more than 1 byte long; DES and Blowfish, for example, have a block size of 8 bytes. AES, the latest addition to the family, has a block size of 16 bytes. The effect of this is that the input data to a cipher that is being used in a blocking mode must be aligned to the block size of that cipher. Truth is, for most of us, the data we wish to encrypt is not always going to be a multiple of the block size of the encryption mechanism we want to use. So while we can find out what the block size is using the `Cipher.getBlockSize()` method and then try to take it into account, the easiest way to deal with this issue is to use padding mechanisms.

PKCS #5/PKCS #7 Padding

PKCS #5 padding was originally developed for block ciphers with a block size of 8 bytes. Later, with the writing of PKCS #7, the authors of the standard specified a broader interpretation of the padding mechanism, which allowed for the padding mechanism to be used for block sizes up to 255 bytes. The PKCS in PKCS #5 and PKCS #7 comes from Public-Key Cryptography Standards that were developed by RSA Security. They are also worth a read, and a list of the most relevant appears in Appendix D.



PKCS #5/#7 Padding with an 8 Byte Block Cipher

Figure 2-1

You can see from Figure 2-1 that the padding mechanism is quite simple; if you need to pad a block of data where the last input block is 3 bytes shorter than the block size of the cipher you are using; you add 3 bytes of value 3 to the data before encrypting it. Then when the data is decrypted, you check the last byte of the last decrypted block of data and remove that many bytes from it. The only shortcoming of this approach is that you must always add the padding, so if the block size of your cipher is 8 bytes and your data is a multiple of 8 bytes in length, you have to add a pad block with 8 bytes with the value 8 to your data before you encrypt it, and as before, remove the extra 8 bytes at the other end when the data is decrypted. The advantage of this approach is that the mechanism is unambiguous.

Try It Out Adding Padding

Fortunately, as a by-product of the use of the factory pattern, the JCE allows you to introduce padding in a manner that makes its effect on the application developer almost invisible. Looking back at the example program again, imagine that you wanted to encrypt and decrypt a hex string which is not block aligned, say, 50 percent longer than the test vector. Here is what the example looks like with PKCS #7 padding introduced and the important changes highlighted:

```
package chapter2;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;


/**
 * Basic symmetric encryption example with padding
 */
public class SimpleSymmetricPaddingExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]        input = new byte[] {
                          0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                          0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                          0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };
        byte[]        keyBytes = new byte[] {
                          0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                          0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                          0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };

        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");

        Cipher        cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");

        System.out.println("input : " + Utils.toHex(input));

        // encryption pass

        cipher.init(Cipher.ENCRYPT_MODE, key);

        byte[] cipherText = new byte[cipher.getOutputSize(input.length)];

        int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

        ctLength += cipher.doFinal(cipherText, ctLength);

        System.out.println("cipher: " + Utils.toHex(cipherText)
                                              + " bytes: " + ctLength);

        // decryption pass

        cipher.init(Cipher.DECRYPT_MODE, key);
```

```
            byte[] plainText = new byte[cipher.getOutputSize(ctLength)];

            int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);

            ptLength += cipher.doFinal(plainText, ptLength);

            System.out.println("plain : " + Utils.toHex(plainText)
                                                    + " bytes: " + ptLength);
        }
    }
```

Run the program and you will see the following output:

```
input : 000102030405060708090a0b0c0d0e0f1011121314151617
cipher: 0060bffe46834bb8da5cf9a61ff220aefa46bbd3578579c0fd331874c7234233 bytes: 32
plain : 000102030405060708090a0b0c0d0e0f10111213141516170000000000000000 bytes: 24
```

Looking through the example, you can see there are not many changes. Looking at the output, you are probably wondering why there are 32 bytes in the `plainText` array when `outLength` is 24. Why are there an extra 8 zero bytes on the end? I will get to that in a minute; first, take a look at the how it works.

How It Works

The key to getting this example to function the way it does is in the string passed in the padding section of the cipher name. Rather than specifying, as you did in the previous Try It Out ("Using AES") :

```
            Cipher        cipher = Cipher.getInstance("AES/ECB/NoPadding", "BC");
```

you replace the padding term in the cipher name, in the above `"NoPadding"`, with the name of the padding you wish to use `"PKCS7Padding"`. This results in the following:

```
            Cipher        cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");
```

The padding has another effect on the code as well. The first thing you will notice is that `Cipher.init()` method is now called before the output arrays are created, and the output array is created by calling `Cipher.getOutputSize()` and passing in the length of the input as an argument. In the case of decryption, this gives you these two lines:

```
            cipher.init(Cipher.DECRYPT_MODE, key);

            byte[] plainText = new byte[cipher.getOutputSize(cipherText.length)];
```

The `Cipher.init()` needs to be first as, if you recall the earlier discussion, a cipher must be initialized before it makes sense to use any of its other methods. In this case, the method you want to use is `Cipher.getOutputSize()`. Note that the JavaDoc for this method specifies that the length returned by `Cipher.getOutputSize()` may be greater than the actual length returned by `Cipher.update()` or `Cipher.doFinal()`, and it often is — especially when decrypting. This is another reason why you need to pay attention to the return values from `Cipher.update()` and `Cipher.doFinal()`. People occasionally make the mistake of thinking this means that there is something broken about the `Cipher` class. If you are wondering about this, remember when a padded message is being decrypted, the `Cipher` class

has no way of knowing how much padding there is *until it decrypts the block with the padding*. You could say it is a fact of life — the best you can do with just the length of the input is guess. Consequently that is all the API offers.

In later examples, you'll use the two-parameter version of `Utils.toHex()` so that only the generated bytes in the output arrays are printed. For the most part, however, you can assume extra bytes will be getting allocated as a result of `Cipher.getOutputSize()`. It is just that they will be ignored, as they should be, by keeping track of the length of the data using the return values from `Cipher.update()` and `Cipher.doFinal()`.

> **Always remember that the estimated size of the output array returned by `Cipher.getOutputSize()` will almost always be larger than the number of bytes produced by the cipher. If you do not take this into account, you will end up with spurious zeros at the end of your data.**

Other Padding Mechanisms

A number of other padding modes are available. The following ones are available in the Bouncy Castle provider. If you are using another provider, you might find some or all of the following in addition to NoPadding and PKCS5Padding and/or PKCS7Padding:

- **ISO10126-2Padding.** A padding mechanism defined in ISO10126-2. The last byte of the padding is the number of pad bytes; the remaining bytes of the padding are made up of random data.
- **ISO7816-4Padding.** A padding mechanism defined in ISO7816-4. The first byte of the padding is the value 0x80; the remaining bytes of the padding are made up of zeros.
- **X9.23Padding.** A padding mechanism defined in X9.23. The last byte of the padding is the number of pad bytes; outside of the last byte, the pad bytes are then either made up of zeros or random data.
- **TBCPadding.** For Trailing Bit Complement padding. If the data ends in a zero bit, the padding will be full of ones; if the data ends in a one bit, the padding will be full of zeros.
- **ZeroBytePadding.** Do not use this one unless you have to deal with a legacy application. It is really only suitable for use with printable ASCII data. In this case the padding is performed by padding out with one or more bytes of zero value. Obviously, if your data might end with bytes of zero value, this padding mechanism will not work very well.

In addition to padding mechanisms that affect the processing of the last block of the data stream, there are cipher modes that affect the processing of each block in the data stream as well. The next section looks at the cipher modes.

Symmetric Block Cipher Modes

Quite a number of modes have been proposed for symmetric block ciphers. This chapter restricts itself to the most well known, but you can find further details on cipher modes by referring to *Applied Cryptography — Second Edition* and *Practical Cryptography*, both of which are listed in Appendix D.

The first one, known as ECB mode, is the mode closest to the actual cipher. The other modes — CBC mode, CTS mode, CTR mode, OFB mode, and CFB mode — are all really constructed on top of ECB mode and attempt to work around problems that can result from using ECB mode directly or because the cipher requires a block of bits at a time to do its job, rather than being able to stream the data.

Let's start with ECB mode.

ECB Mode

ECB, or *Electronic Code Book*, mode describes the use of a symmetric cipher in its rawest form. The problem with ECB mode is that if there are patterns in your data, there will be patterns in your encrypted data as well. A *pattern*, in this case, is any block of bytes that contains the same values as another block of bytes. This is more common than you might imagine, especially if you are processing data that is structured.

Try It Out Ciphertext Patterns in ECB Mode

Try running the following example. The example uses DES not so much as a recommendation, but more because having an 8-byte block size (rather than the 16-byte one AES has) makes it much easier to see the patterns.

```
package chapter2;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;


/**
 * Basic symmetric encryption example with padding and ECB using DES
 */
public class SimpleECBExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]        input = new byte[] {
                              0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                              0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                              0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
        byte[]       keyBytes = new byte[] {
                              0x01, 0x23, 0x45, 0x67,
                              (byte)0x89, (byte)0xab, (byte)0xcd, (byte)0xef };

        SecretKeySpec key = new SecretKeySpec(keyBytes, "DES");

        Cipher        cipher = Cipher.getInstance("DES/ECB/PKCS7Padding", "BC");

        System.out.println("input : " + Utils.toHex(input));

        // encryption pass

        cipher.init(Cipher.ENCRYPT_MODE, key);

        byte[] cipherText = new byte[cipher.getOutputSize(input.length)];
```

```
            int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

            ctLength += cipher.doFinal(cipherText, ctLength);

            System.out.println("cipher: " + Utils.toHex(cipherText, ctLength)
                                                + " bytes: " + ctLength);

            // decryption pass

            cipher.init(Cipher.DECRYPT_MODE, key);

            byte[] plainText = new byte[cipher.getOutputSize(ctLength)];

            int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);

            ptLength += cipher.doFinal(plainText, ptLength);

            System.out.println("plain : " + Utils.toHex(plainText, ptLength)
                                                + " bytes: " + ptLength);
        }
    }
```

When you run this example, you should see the output:

```
input : 000102030405060708090a0b0c0d0e0f0001020304050607
cipher: 3260266c2cf202e28325790654a444d93260266c2cf202e2086f9a1d74c94d4e bytes: 32
plain : 000102030405060708090a0b0c0d0e0f0001020304050607 bytes: 24
```

How It Works

The words "Code Book" really sum up ECB mode. Given a particular block of bytes on input, the cipher performs a set of deterministic calculations, looking up a virtual code book as it were, and returns a particular block of bytes as output. So given the same block of input bytes, you will always get the same block of output bytes. This is how a cipher works in its rawest form.

Notice how the hex string `3260266c2cf202e2` repeats in the ciphertext as the string `001020304050607` does in the input. If you imagine you know nothing about the input data and are looking at the encrypted data and hoping to work out what the input data might contain, that pattern will tell you the input data is repeating. If you already know something about the input data and would like to know more, the pattern might tell you a lot. If the person who is generating the encrypted data is also using the same key repeatedly, a beautiful world might be about to unfold for you if you are the attacker. As for the person doing the encryption, you can see there is a problem.

CBC Mode

CBC, or *Cipher Block Chaining*, mode reduces the likelihood of patterns appearing in the ciphertext by XORing the block of data to be encrypted with the last block of ciphertext produced and then applying the raw cipher to produce the next block of ciphertext.

Try It Out Using CBC Mode

Try the following example:

```
package chapter2;

import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * Basic symmetric encryption example with padding and CBC using DES
 */
public class SimpleCBCExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]          input = new byte[] {
                                0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                                0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                                0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
        byte[]          keyBytes = new byte[] {
                                0x01, 0x23, 0x45, 0x67,
                                (byte)0x89, (byte)0xab, (byte)0xcd, (byte)0xef };
        byte[]          ivBytes = new byte[] {
                                0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00 };

        SecretKeySpec   key = new SecretKeySpec(keyBytes, "DES");
        IvParameterSpec ivSpec = new IvParameterSpec(ivBytes);
        Cipher          cipher = Cipher.getInstance("DES/CBC/PKCS7Padding", "BC");

        System.out.println("input : " + Utils.toHex(input));

        // encryption pass

        cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);

        byte[] cipherText = new byte[cipher.getOutputSize(input.length)];

        int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

        ctLength += cipher.doFinal(cipherText, ctLength);

        System.out.println("cipher: " + Utils.toHex(cipherText, ctLength)
                                            + " bytes: " + ctLength);

        // decryption pass

        cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);

        byte[] plainText = new byte[cipher.getOutputSize(ctLength)];

        int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);
```

```
            ptLength += cipher.doFinal(plainText, ptLength);

            System.out.println("plain : " + Utils.toHex(plainText, ptLength)
                                                + " bytes: " + ptLength);
        }
    }
```

You should see the following output:

```
input : 000102030405060708090a0b0c0d0e0f0001020304050607
cipher: 8a87d41c5d3caead0c21f1b3f12a6cd75424fa086e029e404c89d4c1b9457818 bytes: 32
plain : 000102030405060708090a0b0c0d0e0f0001020304050607 bytes: 24
```

Notice that this time every block of the encrypted output is different, even though you can see that the first and third blocks of the input data are the same. The other item of interest about the ciphertext in this example is that the first block is also different from the first block in the ECB example, despite the fact that they use the same key.

How It Works

You can see from the highlighted changes that we are now passing "CBC" rather than "ECB" to the static `Cipher.getInstance()` method. This explains how I have moved from ECB to CBC mode, but how does that explain the change in the output?

Remember, I said earlier that CBC mode works by XORing the last block of ciphertext produced with the current block of input and then applying the raw cipher. This explains how the first and the third blocks of the ciphertext are now different, as the third block of the ciphertext is now the result of encrypting the XOR of the third block of input with the second block of ciphertext. The question is, what do you do about the first block? At that stage nothing has been encrypted yet.

This is where the `javax.crypto.spec.IvParameterSpec` object comes in. It is used to carry the initialization vector, or IV, and as the name indicates, the `IvParameterSpec` is required, in addition to the key, to initialize the `Cipher` object. It is the initialization vector that provides the initial block of "cipher text" that is XORed with the first block of input.

> **Forgetting to set the IV or setting it to the wrong value is a very common programmer error. The indicator for this error is that the first block of the message will decrypt to garbage, but the rest of the message will appear to decrypt correctly.**

Inline IVs

As you can see, the JCE assumes that the IV will be passed as an out-of-band parameter. Although this is often the case, there is another way of dealing with IVs apart from introducing the IV as an out-of-band parameter to the encryption. In some cases people also write the IV out at the start of the stream and then rely on the receiver to read past it before attempting to reconstruct the message. It's okay to do this, as the IV does not need to be kept secret, but if you are using the JCE, you still need to provide an IV to `Cipher.init()` if you are using a cipher and mode that expects one. Fortunately, this is easy to do as well; in this case you can simply use an IV, which is a block of zeros.

Try It Out Using an Inline IV

Look at the following example:

```
package chapter2;

import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * Symmetric encryption example with padding and CBC using DES
 * with the initialization vector inline.
 */
public class InlineIvCBCExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]          input = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
        byte[]          keyBytes = new byte[] {
                            0x01, 0x23, 0x45, 0x67,
                           (byte)0x89, (byte)0xab, (byte)0xcd, (byte)0xef };
        byte[]          ivBytes = new byte[] {
                           0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00 };

        SecretKeySpec   key = new SecretKeySpec(keyBytes, "DES");
        IvParameterSpec ivSpec = new IvParameterSpec(new byte[8]);
        Cipher          cipher = Cipher.getInstance("DES/CBC/PKCS7Padding", "BC");

        System.out.println("input : " + Utils.toHex(input));

        // encryption pass

        cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);

        byte[] cipherText = new byte[
                           cipher.getOutputSize(ivBytes.length + input.length)];

        int ctLength = cipher.update(ivBytes, 0, ivBytes.length, cipherText, 0);

        ctLength += cipher.update(input, 0, input.length, cipherText, ctLength);

        ctLength += cipher.doFinal(cipherText, ctLength);

        System.out.println("cipher: " + Utils.toHex(cipherText, ctLength)
                                                    + " bytes: " + ctLength);

        // decryption pass

        cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);
```

```
            byte[] buf = new byte[cipher.getOutputSize(ctLength)];

            int bufLength = cipher.update(cipherText, 0, ctLength, buf, 0);

            bufLength += cipher.doFinal(buf, bufLength);

            // remove the iv from the start of the message

            byte[] plainText = new byte[bufLength - ivBytes.length];

            System.arraycopy(buf, ivBytes.length, plainText, 0, plainText.length);

            System.out.println("plain : " + Utils.toHex(plainText, plainText.length)
                                                    + " bytes: " + plainText.length);
        }
    }
```

Run this example and you should see the following output:

```
input : 000102030405060708090a0b0c0d0e0f0001020304050607
cipher: 159fc9af021f30024211a5d7bf88fd0b9e2a82facabb493f39c5a9febe6a659e85039332be5
6f6a4 bytes: 40
plain : 000102030405060708090a0b0c0d0e0f0001020304050607 bytes: 24
```

How It Works

You can see by examining the highlighted pieces of code that there are only two real changes, apart from the use of an `IvParameterSpec` with an array of zero value. First, you now call update twice when you encrypt the message, once to feed in the IV and a second time to feed in the message. Second, you trim the IV block off the start of the plaintext so that you only display the bytes making up the message.

This does save you the trouble of passing the IV out of band with the encrypted message; on the other hand, it makes the encrypted message a block longer, thus increasing the overhead required to send the message. It also complicates the code required to process it.

Creating an IV

The examples in this chapter use a fixed IV. Although this is very useful for demonstrating what is going on, as it makes the output of the examples predictable, producing predictable ciphertext is not something you want in a real-life application. In real life the messages your applications are encrypting are often very similar, and in any case, you often cannot control what will be encrypted generating IVs, dealing with the worst-case scenario appears to be the best approach. A sensible IV should be as random as you can make it and preferably unique to a given message. This gives you two ways of generating an IV: generate a random IV from a random source or generate a pseudorandom IV using some piece of data unique to the message you want to encrypt, such as the message number.

As you can see, there are a few things to think about. If you are interested in a more thorough but still approachable discussion of IV generation, read the discussion on initialization vectors in Chapter 5 of *Practical Cryptography*. (See Appendix D.)

Random IVs

How you generate an IV depends a lot on what the environment is like. If you are thinking about using a random IV, the major consideration is really whether the overhead that is added when you use one is acceptable. If your messages are short and you are sending a lot of them, this can rapidly build up. That being said, if you decide to use a random IV, your best bet is to use a `SecureRandom` object, see that it is seeded appropriately, and generate the bytes you need. You will look at creating a `SecureRandom` object in the next section. For now, it enough to say that generating a random IV will often involve no more than the following:

```
byte[]           ivBytes = new byte[cipherBlockSize];
        random.nextBytes(ivBytes);
IvParameterSpec  ivSpec = new IvParameterSpec(ivBytes);
```

where *`cipherBlockSize`* is the block size of the particular cipher you are using and *`random`* is a `SecureRandom` object.

Alternatively, you can let the `Cipher` object generate an IV for you. This will only happen on encryption—obviously, to decrypt a message, you need to be told the IV. In any case, you could replace the encryption initialization step in the CBC example with

```
        cipher.init(Cipher.ENCRYPT_MODE, key);
        IvParameterSpec ivSpec = new IvParameterSpec(cipher.getIV());
```

and take advantage of the `Cipher` object's ability to generate a random IV. Of course, in real life, you would probably just retrieve the raw bytes using `Cipher.getIV()` and pass them along with the message to the receiver, who would then create an `IvParameterSpec`.

Creating a SecureRandom Object

An object based on the `java.security.SecureRandom` class can be as simple or as complicated to create as you want. Initially, they were created solely using constructors and a Sun provider implementation based on using a SHA-1 hash. In earlier versions of the JDK, the default initialization of the class caused major performance issues. These days there is support for the creation of `SecureRandom` objects through the factory pattern, meaning that the JCA provider implementers can provide their own, and the default `SecureRandom` implementation will take advantage of hardware support for random number generation as well.

The upshot is that for the most part, `new SecureRandom()` will probably do the right thing by you.

If you are using an older version of the JDK and the default seeding mechanism is destroying performance by causing a substantial delay when the first `SecureRandom` is created, you can prevent the default initialization from taking place by using the `SecureRandom` constructor that takes a byte array. Just make sure you add enough *seed material,* in the shape of random timings from mouse events, network events, or input from the keyboard or any other source available to you, to the `SecureRandom` object you have created to make sure you are getting a good-quality random seed. Things like just the process identifier and system time are not enough. You need to have enough sources of entropy to make sure the initial state of your `SecureRandom` is not easily guessable. Covering all the possible ways to collect seed material is a discussion that is not really appropriate to this book; however, if you are interested in looking into this further, you might want to look at RFC 1750 and also Chapter 10 of *Practical Cryptography* (see Appendix D for more information about this book).

Pseudorandom IVs

Most systems incorporate some idea of message numbering, if for no other reason than to avoid replay attacks. In any case, I mention message numbers because they are generally unique to the message across the system, but the general idea is to find something that travels along with the message that occurs only once. Another name for such a value is a *nonce,* which is short for *number used once.*

Having found a suitable nonce value, you can generate an IV for your message by using the nonce as a seed to some other function that will generate the bytes you need for the IV. As you are after a block size for the cipher worth of bytes and you would like it to be unique, just encrypting the nonce with the cipher will work nicely. In this case you only need to use ECB mode; however, you can avoid creation of an extra `Cipher` object by using the CBC cipher with a zero IV — which is exactly the same.

Try It Out — Using an IV Based on a Nonce

Try the following example:

```
package chapter2;

import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * CBC using DES with an IV based on a nonce. In this
 * case a hypothetical message number.
 */
public class NonceIvCBCExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]        input = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
        byte[]        keyBytes = new byte[] {
                            0x01, 0x23, 0x45, 0x67,
                            (byte)0x89, (byte)0xab, (byte)0xcd, (byte)0xef };
        byte[]        msgNumber = new byte[] {
                            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

        IvParameterSpec zeroIv = new IvParameterSpec(new byte[8]);

        SecretKeySpec   key = new SecretKeySpec(keyBytes, "DES");

        Cipher          cipher = Cipher.getInstance("DES/CBC/PKCS7Padding", "BC");


        System.out.println("input : " + Utils.toHex(input));

        // encryption pass
```

```
            // generate IV

            cipher.init(Cipher.ENCRYPT_MODE, key, zeroIv);

            IvParameterSpec encryptionIv = new IvParameterSpec(
                                                    cipher.doFinal(msgNumber), 0, 8);

            // encrypt message

            cipher.init(Cipher.ENCRYPT_MODE, key, encryptionIv);

            byte[] cipherText = new byte[cipher.getOutputSize(input.length)];

            int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

            ctLength += cipher.doFinal(cipherText, ctLength);

            System.out.println("cipher: " + Utils.toHex(cipherText, ctLength)
                                                     + " bytes: " + ctLength);

            // decryption pass

            // generate IV

            cipher.init(Cipher.ENCRYPT_MODE, key, zeroIv);

            IvParameterSpec decryptionIv = new IvParameterSpec(
                                                    cipher.doFinal(msgNumber), 0, 8);

            // decrypt message

            cipher.init(Cipher.DECRYPT_MODE, key, decryptionIv);

            byte[] plainText = new byte[cipher.getOutputSize(ctLength)];

            int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);

            ptLength += cipher.doFinal(plainText, ptLength);

            System.out.println("plain : " + Utils.toHex(plainText, ptLength)
                                                     + " bytes: " + ptLength);
        }
    }
```

Run this example and you will now see the following output:

```
input : 000102030405060708090a0b0c0d0e0f0001020304050607
cipher: eb913126049ccdea00f2d86fda94a02fd72e0914fd361400d909f45f73058fc3 bytes: 32
plain : 000102030405060708090a0b0c0d0e0f0001020304050607 bytes: 24
```

As you can see, the ciphertext in the output has now changed substantially as a consequence of the change in IV.

How It Works

Looking at the code, there is only really one major change from the original CBC example: the creation of and handling of the IV. Note that in both cases when the IV is calculated from the message number, the `Cipher` object is initialized for encryption. The reason is that you are not so much encrypting the message number (stored in the array `msgNumber`) as using the cipher's encryption mode to calculate an IV from the message number.

This example also introduces one of the convenience methods on `Cipher`, a `Cipher.doFinal()`, which does the full processing on the input array and produces the resulting ciphertext, complete with padding. The presence of the padding is the reason why you specify that you only want the first 8 bytes of the ciphertext used in the creation of the `IvParameterSpec` object; otherwise, the IV will be two blocks, rather than the required one.

A Look at Cipher Parameter Objects

You have seen already how an IV can be passed into `Cipher.init()` using an `IvParameterSpec` object. You may have noticed that `Cipher.init()` can also take `AlgorithmParameters` objects. Likewise, just as there is a `Cipher.getIV()`, there is also a `Cipher.getParameters()` method.

> **`Cipher.getIV()` and `Cipher.getParameters()` should only be called after `Cipher.init()` has been called on the cipher of interest.**

At this point it would be worth looking briefly at what the difference is between parameter objects that end in the word `Spec` and those that do not. As a rule, in the JCE, objects ending in the word `Spec` are just value objects. Although these are useful in their own right, there are also situations where you need to be able to retrieve the parameters of a `Cipher`, or some other processing class, not as a value object but as an object that will produce an encoded version suitable for transmission to someone else, or for preservation in a platform-independent manner.

The AlgorithmParameters Class

The `AlgorithmParameters` objects serve this purpose and contain not just the values for the parameters but also expose methods such as `AlgorithmParameters.getEncoded()`, which allow the parameters to be exported in a platform-independent manner. The most common encoding method that `AlgorithmParameters` objects use is one of the binary encodings associated with ASN.1, which is discussed in Chapter 5. The `AlgorithmParameters` class also has a method on it, `AlgorithmParameters.getParameterSpec()`, which enables you to recover the value object associated with the parameters contained in the `AlgorithmParameters` object.

Consequently, calling `Cipher.getParameters()` will, amongst other things, return the IV, but in an object that can be used to generate an encoded IV, suitable for export.

CTS Mode: A Special Case of CBC

You could make use of CTS, or Cipher Text Stealing, mode in the previous example by replacing the `Cipher.getInstance()` call with

```
Cipher          cipher = Cipher.getInstance("DES/CTS/NoPadding", "BC");
```

CTS is defined in RFC 2040 and combines the use of CBC mode with some additional XOR operations on the final encrypted block of the data being processed to produce encrypted data that is the same length as the input data. In some ways, it is almost a padding mechanism more than a mode, and as it is based around CBC mode, it still requires the data to be processed in discrete blocks. If you want to be able to escape from having to process the data in blocks altogether, you need to use one of the streaming block modes, which are covered next.

Streaming Symmetric Block Cipher Modes

Both ECB and CBC mode use the underlying cipher in its most basic way, as an engine that takes in a block of data and outputs a block of data. Of course, the result of doing this was that in situations where you did not have data that was a multiple of the block size in length, you needed to use padding. Although this is an improvement on the situation, it would be useful to be able to use a regular block cipher in a manner that allows you to produce encrypted messages that are the same length as the initial unencrypted messages without having to resort to the kind of shenanigans that take place in CTS. Streaming block cipher modes allow you to use a block cipher in this way.

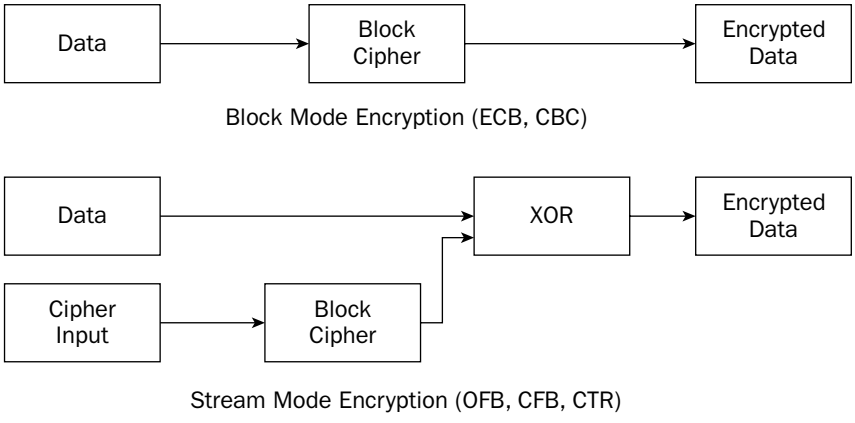


Figure 2-2

Look at Figure 2-2 and you will see how the streaming is possible. Unlike ECB and CBC modes, the three following modes work by producing a stream of bits that is then XORed with the plaintext. One major thing you need to be careful of: Reusing an IV and a key together is fatal to the security of the encryption. As mentioned previously, you should not do this with CBC mode either, but your exposure, if you do so, will normally be limited. In the case of the stream modes, your exposure from reusing the initialization vector will be total. The reasons for this vary slightly depending on which mode you are using, but the principle remains the same.

CTR Mode

Also known as SIC (*Segmented Integer Counter*) mode. CTR, or *Counter* mode, has been around for quite a while but has finally been standardized by NIST in SP 800-38a and in RFC 3686.

Try It Out CTR Mode

Consider the following example:

```
package chapter2;

import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * Basic symmetric encryption example with CTR using DES
 */
public class SimpleCTRExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]          input = new byte[] {
                                0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                                0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                                0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };
        byte[]          keyBytes = new byte[] {
                                0x01, 0x23, 0x45, 0x67,
                                (byte)0x89, (byte)0xab, (byte)0xcd, (byte)0xef };
        byte[]          ivBytes = new byte[] {
                                 0x00, 0x01, 0x02, 0x03, 0x00, 0x00, 0x00, 0x01 };

        SecretKeySpec   key = new SecretKeySpec(keyBytes, "DES");
        IvParameterSpec ivSpec = new IvParameterSpec(ivBytes);
        Cipher          cipher = Cipher.getInstance("DES/CTR/NoPadding", "BC");

        System.out.println("input : " + Utils.toHex(input));

        // encryption pass

        cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);

        byte[] cipherText = new byte[cipher.getOutputSize(input.length)];

        int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

        ctLength += cipher.doFinal(cipherText, ctLength);

        System.out.println("cipher: " + Utils.toHex(cipherText, ctLength)
                                            + " bytes: " + ctLength);

        // decryption pass

        cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);

        byte[] plainText = new byte[cipher.getOutputSize(ctLength)];

        int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);
```

```
            ptLength += cipher.doFinal(plainText, ptLength);

            System.out.println("plain : " + Utils.toHex(plainText, ptLength)
                                                    + " bytes: " + ptLength);
        }
    }
```

Running the example, you should see the following output:

```
input : 000102030405060708090a0b0c0d0e0f00010203040506
cipher: 61a1f886ff9bc709dd37cd9ce33adc6ff9ab110e46f387 bytes: 23
plain : 000102030405060708090a0b0c0d0e0f00010203040506 bytes: 23
```

As you can see, the `Cipher` object has produced ciphertext that is the same length as the input data.

How It Works

From a coding point of view, because of the benefit of the factory pattern, the only major change from the CBC example is that you have called `Cipher.getInstance()` with `CTR` rather than `CBC` specified in the mode position, and as you are using a streaming mode, `NoPadding` rather than `PKCS7Padding` giving a specification string of `"DES/CTR/NoPadding"`. You do not need to specify any padding because the mode allows you to work with any length of data.

Note also that the IV ends in 3 zero bytes and a one byte. In this case, it is a way of telling yourself that you should limit your processing to data that is no more than 2^{35} bytes (2^{32} times the block size). After that, the counter will go back to zero and begin cycling at the next block. You can see that encrypting two messages with the same IV and the same key will result in encrypting both messages with the same stream of bits. How many messages you can encrypt with a single key depends on how you treat the first four bytes of the IV. In a situation like this, you might divide the four bytes in half and use the first two for the message number and the second two for random data. This would allow you to process 2^{16} messages before recycling keys.

If you follow these rules, CTR mode works very well. There are three nice things about CTR mode: It is a stream mode, so is very easy to work with as you do not have to worry about padding; it allows for random access to the encrypted data, as you just need to know the counter value for a particular block; and finally, the areas where you can get into trouble using CTR mode are obvious from the design for construction of the IV. Given a particular method of constructing an IV, it is easy to see how large a message you can encrypt and how many messages you can process before you have to change keys. You can be certain of this, as you know that the cipher will be producing a different block for each increment of the counter until the counter begins to cycle. You will not have any surprises — in cryptography, this is a good thing.

OFB Mode

You can make use of OFB, or *Output Feedback*, mode in the previous example by replacing the `Cipher.getInstance()` to create the CTR cipher with the following:

```
            Cipher          cipher = Cipher.getInstance("DES/OFB/NoPadding", "BC");
```

Like CTR mode, OFB mode works by using the raw block cipher to produce a stream of pseudorandom bits, which are then XORed with the input message to produce the encrypted message. The actual input

message is never used. With OFB mode, rather than considering part of the IV to be a counter, you just load the IV into a state array, encrypt the state array, and save the result back to the state array, using the bits you generated to XOR with the next block of input and generate the ciphertext.

You might also see the following:

```
            Cipher          cipher = Cipher.getInstance("DES/OFB8/NoPadding", "BC");
```

If this is the case, the OFB mode is set so that the cipher behaves like it has a block size of 8 bits. Virtual block sizes of 16, 24, 32, and so on are also possible. Do not do this unless you do so for reasons of compatibility. Security analysis of OFB mode has shown that it should be used only with an apparent block size that is the same as the block size of the underlying cipher.

The next biggest problem with OFB mode is that if the repetition of encrypting the state initialized by the IV leads to another state value that has occurred before, the value of the state will simply become a repetition of what occurred previously. You have to process a lot of data, and be unlucky, for this to be a problem. As a general rule in cryptography, it is better to avoid anything that involves the word *luck* where possible. Current wisdom is to use CTR instead of OFB, as it gives you more control over the key stream.

CFB Mode

CFB, or *Cipher Feedback*, is one you will encounter a lot. Its most widespread application is probably in the OpenPGP message format, described in RFC 2440, where it used as the mode of choice.

You can make use of CFB mode in the previous example by replacing the `Cipher.getInstance()` to create the CTR cipher with the following:

```
            Cipher          cipher = Cipher.getInstance("DES/CFB/NoPadding", "BC");
```

Like OFB mode and CTR mode, CFB mode produces a stream of pseudorandom bits that are then used to encrypt the input. Unlike the others, CFB mode uses the plaintext as part of the process of generating the stream of bits. In this case, CFB starts with the IV, encrypts it using the raw cipher and saves it in a state array. As you encrypt a block of data, you XOR it with the state array to get the ciphertext and store the resulting ciphertext back in our state array.

Like OFB you can also use CFB mode in the following manner:

```
            Cipher          cipher = Cipher.getInstance("DES/CFB8/NoPadding", "BC");
```

This actually changes the way the plaintext gets fed into the state array. Using the 8-bit mode described previously, after each encryption step, the bytes in the state array will be shifted left 8 bits, and the byte of ciphertext that was produced will be added to the end (something very similar happens in OFB mode as well, but since you should avoid using OFB mode unless you have to, you probably do not need to worry about it). Obviously one downside here is that the smaller the block size dictated by the mode, the more encryption operations there are. In the case of AES where you have a 16-byte block, using CFB in a 16-bit mode will mean you will have eight times as many encryption operations. The downside of using CFB in full block mode is you then risk running into the same problem as OFB with the bit stream starting to repeat. CFB mode does have some interesting properties in respect to dealing with synchronization errors, so for some applications it is probably a contender. It depends on what you are trying to do.

Symmetric Stream Ciphers

For the purposes here, stream ciphers are basically just ciphers that, by design, behave like block ciphers using the streaming modes. Once again, the idea is for the cipher to create a stream of bits that are then XORed with the plaintext to produce the ciphertext. From the point of view of using stream ciphers in the JCE, you will not notice much difference, other than in the creation of the `Cipher` objects. Stream ciphers do not have modes or require padding — they will always produce output the same length as the input. The result is that only the name of the algorithm is required.

Try It Out Using the ARC4 Stream Cipher

Here is a simple example using a 128-bit key, for what is probably the most widely used stream cipher on the net — ARC4, apparently based on RSA Security's RC4 cipher. Note that `Cipher.getInstance()` is just passed the name of the algorithm.

```
package chapter2;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;


/**
 * Basic stream cipher example
 */
public class SimpleStreamExample
{
    public static void main(
        String[]    args)
        throws Exception
    {
        byte[]        input = new byte[] {
                            0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
                            (byte)0x88, (byte)0x99, (byte)0xaa, (byte)0xbb,
                            (byte)0xcc, (byte)0xdd, (byte)0xee, (byte)0xff };
        byte[]        keyBytes = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };

        SecretKeySpec key = new SecretKeySpec(keyBytes, "ARC4");

        Cipher        cipher = Cipher.getInstance("ARC4", "BC");


        System.out.println("input text : " + Utils.toHex(input));

        // encryption pass

        byte[] cipherText = new byte[input.length];

        cipher.init(Cipher.ENCRYPT_MODE, key);

        int ctLength = cipher.update(input, 0, input.length, cipherText, 0);
```

```
            ctLength += cipher.doFinal(cipherText, ctLength);

            System.out.println("cipher text: " + Utils.toHex(cipherText)
                                                    + " bytes: " + ctLength);

            // decryption pass

            byte[] plainText = new byte[ctLength];

            cipher.init(Cipher.DECRYPT_MODE, key);

            int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);

            ptLength += cipher.doFinal(plainText, ptLength);

            System.out.println("plain text : " + Utils.toHex(plainText)
                                                    + " bytes: " + ptLength);
        }
    }
```

Running this example should produce the following output:

```
input text : 00112233445566778899aabbccddeeff
cipher text: e98d62ca03b77fbb8e423d7dc200c4b0 bytes: 16
plain text : 00112233445566778899aabbccddeeff bytes: 16
```

As you can see, the ciphertext is the same length as the input text.

How It Works

This example is pretty well the same as any other symmetric cipher example. The only real difference is that you have not specified a mode or a padding, as, in this case, none is required.

One further note on stream ciphers and the JCE: Block size is not really relevant; consequently, the JCE allows a stream cipher to present itself in a manner that allows `Cipher.getBlockSize()` to return 0. If you are trying to write a general-purpose application that will work with any symmetric cipher in the JCE, make sure you do not assume that the return value of `Cipher.getBlockSize()` will always be nonzero.

Generating Random Keys

Up until now, you have been relying on the `SecretKeySpec` class as the object used to create keys for passing into `Cipher.init()`. Looking at the generation of random initialization vectors, you could imagine that one way for generating keys would simply be to generate an array of random bytes and then pass that to a `SecretKeySpec`. Another way, which is preferred, is to use `javax.crypto.KeyGenerator` class.

Try It Out — Random Symmetric Key Generation

Look at the following example, which is built around AES in CTR mode, rather than DES as you saw in the previous Try It Out ("CTR Mode"), and try running it a few times.

```
package chapter2;

import java.security.Key;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * Basic example using the KeyGenerator class and
 * showing how to create a SecretKeySpec from an encoded key.
 */
public class KeyGeneratorExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]          input = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
        byte[]          ivBytes = new byte[] {
                            0x00, 0x00, 0x00, 0x01, 0x04, 0x05, 0x06, 0x07,
                            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 };

        Cipher          cipher = Cipher.getInstance("AES/CTR/NoPadding", "BC");
        KeyGenerator    generator = KeyGenerator.getInstance("AES", "BC");

        generator.init(192);

        Key encryptionKey = generator.generateKey();

        System.out.println("key   : " + Utils.toHex(encryptionKey.getEncoded()));

        System.out.println("input : " + Utils.toHex(input));

        // encryption pass

        cipher.init(Cipher.ENCRYPT_MODE, encryptionKey,
                                            new IvParameterSpec(ivBytes));

        byte[] cipherText = new byte[cipher.getOutputSize(input.length)];

        int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

        ctLength += cipher.doFinal(cipherText, ctLength);

        // create our decryption key using information
        // extracted from the encryption key

        Key    decryptionKey = new SecretKeySpec(
                        encryptionKey.getEncoded(), encryptionKey.getAlgorithm());

        cipher.init(Cipher.DECRYPT_MODE, decryptionKey,
                                            new IvParameterSpec(ivBytes));

        byte[] plainText = new byte[cipher.getOutputSize(ctLength)];
```

```
            int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);

            ptLength += cipher.doFinal(plainText, ptLength);

            System.out.println("plain : " + Utils.toHex(plainText, ptLength)
                                                    + " bytes: " + ptLength);
        }
    }
```

You should find that while the key keeps changing, the input and the decrypted plaintext stay the same, indicating that the generated key has been successfully converted to a `SecretKeySpec`.

How It Works

Looking at the example, there are a few things you will notice. The IV is now 16 bytes as opposed to 8 bytes, reflecting the larger block size AES has over DES. Next, you have created a `javax.crypto.KeyGenerator` class, initialized it, and used it to create a `java.security.Key` object. The Key object is then used to initialize the `Cipher` object and encrypt the data, producing the ciphertext. Finally, you can see that before decryption, a `SecretKeySpec` is created by using `Key.getEncoded()` and `Key.getAlgorithmName()`, and you then initialize the `Cipher` object with the `SecretKeySpec` as you would normally do and decrypt the ciphertext.

The Key Interface

The `javax.security.Key` interface is the base interface implemented by all objects that can be used as cryptographic keys, including `SecretKeySpec`. It has three methods, all of which you will look at here, although you have only needed two of them so far.

Key.getAlgorithm()

This returns the name of the algorithm that the key is for. In the case of the example, this would simply be the string `"AES"`, which is exactly what is required for the second parameter of `SecretKeySpec`.

Key.getEncoded()

In the case of a symmetric key, this just returns the bytes making up the key material. As you will see in Chapter 4, it gets a little more complicated with asymmetric keys. In the case of the previous example, though, you can see that the result of `Key.getEncoded()` can be passed to `SecretKeySpec` in the same way a byte array can.

Key.getFormat()

This returns the format of the byte array returned by `Key.getEncoded()`. As you get further into the JCE, you'll see that this can return a variety of values, but for keys used for symmetric algorithms, the name of the format returned is normally the string `"RAW"`, indicating that `Key.getEncoded()` returns the bytes making up the key material without any additional packaging.

The KeyGenerator Class

The `javax.crypto.KeyGenerator` class is used to generate keys for use with symmetric encryption algorithms.

KeyGenerator.getInstance()

`KeyGenerator`, like `Cipher`, uses a factory pattern for creating instance objects. Unlike the `Cipher` class, there is no mode or padding to specify, so no additional syntax to know. Just the name of the algorithm and the provider is sufficient, as in:

```
KeyGenerator    generator = KeyGenerator.getInstance("AES", "BC");
```

As with any other `getInstance()` method, you can leave the provider name off and the Java runtime will return the first `KeyGenerator` for the passed-in algorithm name that it finds. Remember, you may inadvertently mix incompatible providers if you do this.

KeyGenerator.init()

At a minimum, `KeyGenerator` objects should be given at least the key size to be generated; otherwise, it is largely up to the internals of the generator what size key you get back. You can also pass the `init()` method of a `KeyGenerator` a source of randomness in the shape of a `java.security.SecureRandom` object you have created and seeded yourself. If you do not pass in a source of randomness, the `KeyGenerator` will generate one internally.

There is also an `init()` method that takes an `AlgorithmParameterSpec` object. Whether you wind up using that depends on what algorithm you are using. To date I have found, in the case of symmetric key generation, using `init()` with a key size and a source of randomness has been enough.

KeyGenerator.generateKey()

Returns a key generated according the parameters passed in by `KeyGenerator.init()`. If you look at the JavaDoc for this method you will see that it really returns a `javax.crypto.SecretKey` object. `SecretKey` is an empty interface that extends `Key` to provide type safety for symmetric key objects.

Password-Based Encryption

So far, you have looked at specifying keys by hand and generating them by using random numbers. Another method for generating keys is to take some human-friendly data like a password, process it using some function, or set of functions, and produce something suitable for use with a symmetric cipher. Encryption carried out in this fashion is known as *password-based encryption,* or PBE.

The most widespread PBE mechanisms are published in PKCS #5 and PKCS #12. There is also a PBE scheme that can be used with S/MIME, which is published in RFC 3211. In the latter case, the key generated from the password is used to generate another key.

This brings up an important point. Outside of dealing with the implementation issues, any PBE scheme must make sure that when people choose passwords, they make sense, from a security point of view, with what is being protected. Passwords are very convenient for people to use. Having said that, left to our own devices, we do have a tendency to take advantage of as much of the convenience as we can. If you are using AES and generating 256-bit keys using a PBE scheme, while at the same time allowing your users to use the password "a," you would not be entirely honest if you claim you are taking full advantage of 256-bit AES.

PBE key generation mechanisms do employ methods to prevent trivial attacks, but a determined user can easily thwart them by choosing inappropriate passwords. If you use PBE, make sure you have a policy on passwords that's appropriate to the security your application requires and that the policy is enforced.

Basic PBE

PBE mechanisms are based on cryptographic hashing mechanisms. In essence, a password and a *salt*, which is just random data, is fed in some fashion into a mixing function based around a secure hash and the function is applied the number of times dictated by an *iteration count*. Once the mixing is complete, the resulting byte stream coming out of the PBE mechanism is used to create the key for the cipher to be used and possibly an initialization vector as well.

Most mixing functions are built on top of message digests, which I will cover in more detail in Chapter 3, but you can also find out more about the internals of them by looking at the documents for the standards referred to previously. If you look at Figure 2-3, you will realize that the salt and the iteration count must somehow be stored with the encrypted data so that the original data can be recovered later. Briefly the details about the roles the various inputs play are as follows.

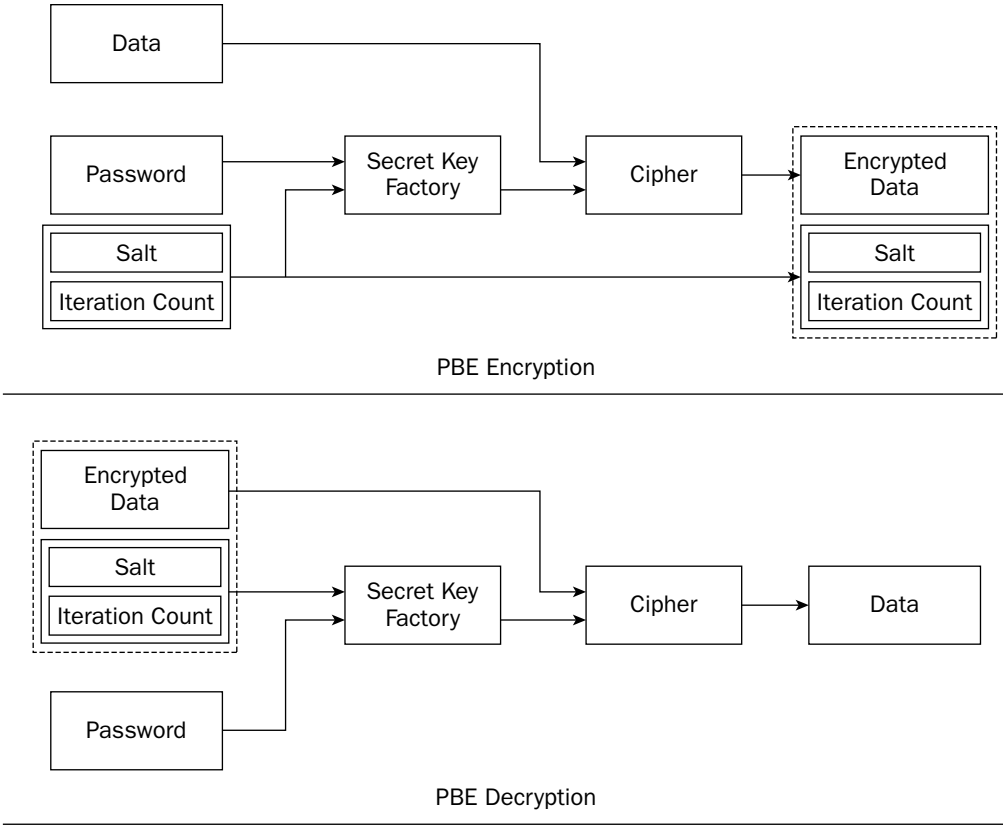


Figure 2-3

The Password

As you have already probably realized, this is the bit that must not only be kept secret, but be at least as hard to guess as the security of your application requires. How much bandwidth you get out of a password in Java depends on the PBE scheme you are using. Most, like PKCS #5, only consider characters in the ASCII range — that is, you will only get the bottom 8 bits of each Java character being mixed into the function. If you are using PKCS #12-based mechanisms, you will be able to utilize the full 16 bits of each Java character.

The Salt

As you can see from Figure 2-3, the salt is a public value — as in you should assume an attacker can find it. The reason for the salt is that by adding a string of random bytes to the password, the same password can be used as a source for a large number of different keys. This is useful because it forces attackers to perform the key generation calculation every time they wish to try a password, for every piece of encrypted data they wish to attack.

If you can, make the salt at least as large as the block size of the function used to process the password. Usually the block size of the function is the same as that of the underlying message digest used by it.

The Iteration Count

As you can see from Figure 2-3, the iteration count is also a public value. The sole purpose of the iteration count is to increase the computation time required to convert a password to a key. For example, imagine someone is trying to launch an attack on data that has been encrypted using PBE by using a dictionary of common words, phrases, and names — more commonly referred to as a *dictionary attack*. If the PBE mechanism has been used with an iteration count of 1,000 rather than 1, it will require 1,000 times more processing to calculate a key from a password.

Make the iteration count as large as you can comfortably. Users usually will cope if an authentication process takes a second or two, and you will be making life a lot harder for someone trying a dictionary attack.

PBE in the JCE

So how does this work in the JCE? There are actually two mechanisms for dealing with PBE in Java, one of which should work across any compliant provider and is based on the `PBEParameterSpec` class. The other mechanism, which is preferred, was introduced in JDK 1.4 and gives the provider more flexibility as to how the key gets generated. You will see the advantage of this a bit later, but which mechanism you choose to use will also depend on whether supporting earlier versions of the JCE is important to you. If you are using the JCE 1.2.2, or a clean room version of the same, the older method is the one you should use.

The following examples have been written for Triple-DES, also referred to as DESede. DESede, a contraction of DES-Encrypt-Decrypt-Encrypt, is derived from the fact that Triple-DES is a three-step single DES process involving first encrypting the input block using single DES with one key, then decrypting the input block using single DES with another key, then encrypting the block again with either the first key used (Two-Key Triple-DES) or a completely different key (Three-Key Triple-DES). At this writing, standards for PBE mechanisms for AES have not yet being established, so I recommend using either Two- or Three-Key Triple-DES.

The other point of interest is the naming convention used for PBE in the JCE. PBE mechanisms are named using the rule PBEwith<*function*>And<*cipher*> where *function* is the algorithm used to support the mechanism that generates the key and *cipher* is the underlying cipher used for the encryption. In the following example, the function used is based on SHA-1 — a message digest, or cryptographic hash. You will look at message digests in Chapter 3, but for now it is enough to know that SHA-1 provides a useful mechanism for mixing bits together in a manner that is not predictable but still deterministic.

Take a look at the original method for using PBE first.

Try It Out PBE Using PBEParameterSpec

Consider the following example; it uses a regular cipher to encrypt the input data and then uses a PBE cipher to decrypt it. In this case, the password is used to create a `javax.crypto.spec.PBEKeySpec`, which is converted to key material by a `javax.crypto.SecretKeyFactory`. The salt and the iteration count are then passed in with the processed key using a `javax.crypto.spec.PBEParameterSpec`.

```
package chapter2;

import java.security.Key;

import javax.crypto.Cipher;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.PBEParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * Example of using PBE with a PBEParameterSpec
 */
public class PBEWithParamsExample
{
    public static void main(String[] args) throws Exception
    {
        byte[]          input = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
        byte[]          keyBytes = new byte[] {
                            0x73, 0x2f, 0x2d, 0x33, (byte)0xc8, 0x01, 0x73,
                            0x2b, 0x72, 0x06, 0x75, 0x6c, (byte)0xbd, 0x44,
                            (byte)0xf9, (byte)0xc1, (byte)0xc1, 0x03, (byte)0xdd,
                            (byte)0xd9, 0x7c, 0x7c, (byte)0xbe, (byte)0x8e };
        byte[]          ivBytes = new byte[] {
                            (byte)0xb0, 0x7b, (byte)0xf5, 0x22, (byte)0xc8,
                            (byte)0xd6, 0x08, (byte)0xb8 };

        // encrypt the data using precalculated keys

        Cipher cEnc = Cipher.getInstance("DESede/CBC/PKCS7Padding", "BC");

        cEnc.init(Cipher.ENCRYPT_MODE,
            new SecretKeySpec(keyBytes, "DESede"),
```

```
            new IvParameterSpec(ivBytes));

        byte[] out = cEnc.doFinal(input);

        // decrypt the data using PBE

        char[]           password = "password".toCharArray();
        byte[]           salt = new byte[] {
                             0x7d, 0x60, 0x43, 0x5f,
                             0x02, (byte)0xe9, (byte)0xe0, (byte)0xae };
        int              iterationCount = 2048;
        PBEKeySpec       pbeSpec = new PBEKeySpec(password);
        SecretKeyFactory keyFact =
                  SecretKeyFactory.getInstance("PBEWithSHAAnd3KeyTripleDES", "BC");

        Cipher cDec = Cipher.getInstance("PBEWithSHAAnd3KeyTripleDES","BC");
        Key    sKey = keyFact.generateSecret(pbeSpec);

        cDec.init(Cipher.DECRYPT_MODE,
                            sKey, new PBEParameterSpec(salt, iterationCount));

        System.out.println("cipher : " + Utils.toHex(out));
        System.out.println("gen key: " + Utils.toHex(sKey.getEncoded()));
        System.out.println("gen iv : " + Utils.toHex(cDec.getIV()));
        System.out.println("plain  : " + Utils.toHex(cDec.doFinal(out)));
    }
}
```

If you run this example, you will see the following output:

```
cipher : a7b955896f750665ba71eb50ac3071d9832a8b02760c600bf619a75a0697c87c
gen key: 007000610073007300770006f007200640000
gen iv : b07bf522c8d608b8
plain  : 000102030405060708090a0b0c0d0e0f0001020304050607
```

The line labeled `gen iv` gives you the real IV used, and as you would expect, it is the same as the one used in the DESede encryption step. However, if you look at the generated key, labeled `gen key`, it is the password broken down into bytes at 2 bytes per character with 2 zero bytes added to the end. As mentioned earlier, this is the method used by PKCS #12 to convert a password into bytes so that it can be fed into the key generation function.

How It Works

When the preprocessed key is passed to the `Cipher.init()` method, the extra information in the `PBEParameterSpec` object is used and the password is mixed into a password generation function and used to create the actual key used for encryption. As you will see in the next section, the actual key generated bears no apparent resemblance to the password that was used to create it.

This need for the `PBEParameterSpec` can be unfortunate because it means, in the event you need to pass it on to an outside application, you have no way of working out what the actual key used by the PBE is. The newer method of using `PBEKeySpec` addresses this drawback, but first it would be worth having a look at the classes introduced by the example in more detail.

The PBEParameterSpec Class

The `javax.crypto.spec.PBEParameterSpec` class is available as a carrier for the salt and the iteration count so that they can be passed to the `Cipher.init()` method. As indicated next, you will see it used in any code that was written for versions of the JCE prior to JDK 1.4, or more recent versions where the provider has not yet caught up with the changes in the `PBEKeySpec`.

The PBEKeySpec Class

Originally `javax.crypto.spec.PBEKeySpec` was just a holder for the password. Since JDK 1.4, it is also able to carry the salt and the iteration count if required for the key generation mechanism. Depending on what provider you are using, the newer version of `PBEKeySpec` obsoletes the `PBEParameterSpec` class.

The SecretKeyFactory Class

Having created a `PBEKeySpec` that contains the password, and possibly the other key generation parameters as well, you need some way of getting the provider you are using to convert the specification into a `Key` object suitable for use with one of the provider's ciphers. The `javax.crypto.SecretKeyFactory` class is provided for this purpose.

Like the other JCE classes, the `SecretKeyFactory` is created using a factory pattern based on the `getInstance()` method, which behaves in exactly the same way as other JCA/JCE `getInstance()` methods. It is a fairly straightforward class with methods on it to make it possible to translate secret keys from one provider to another, generate key specifications from `SecretKey` objects, and take key specifications and convert them into keys. You will look at the first two capabilities later, but in the case of PBE, it is this last capability you are interested in. As you will see in the examples, the `SecretKeyFactory` allows you to reduce the conversion of the `PBEKeySpec` into a `SecretKey` object suitable for passing to `Cipher.init()` to a single method call — `SecretKeyFactory.generateSecret()`.

Try It Out PBE Based Solely on PBEKeySpec

Having seen the original method for handling PBE, you now look at a more up-to-date one. As you can see here, the changes required in the example are both very minor, but there is a major difference in what is returned from the `SecretKeyFactory.generateSecret()` method. Look at the following:

```
package chapter2;

import java.security.Key;

import javax.crypto.Cipher;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * Example of using PBE without using a PBEParameterSpec
 */
public class PBEWithoutParamsExample
{
    public static void main(String[] args) throws Exception
    {
```

```
            byte[]          input = new byte[] {
                                0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                                0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                                0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
            byte[]          keyBytes = new byte[] {
                                0x73, 0x2f, 0x2d, 0x33, (byte)0xc8, 0x01, 0x73,
                                0x2b, 0x72, 0x06, 0x75, 0x6c, (byte)0xbd, 0x44,
                                (byte)0xf9, (byte)0xc1, (byte)0xc1, 0x03, (byte)0xdd,
                                (byte)0xd9, 0x7c, 0x7c, (byte)0xbe, (byte)0x8e };
            byte[]          ivBytes = new byte[] {
                                (byte)0xb0, 0x7b, (byte)0xf5, 0x22, (byte)0xc8,
                                (byte)0xd6, 0x08, (byte)0xb8 };

            // encrypt the data using precalculated keys

            Cipher cEnc = Cipher.getInstance("DESede/CBC/PKCS7Padding", "BC");

            cEnc.init(Cipher.ENCRYPT_MODE,
                new SecretKeySpec(keyBytes, "DESede"),
                new IvParameterSpec(ivBytes));

            byte[] out = cEnc.doFinal(input);

            // decrypt the data using PBE

            char[]              password = "password".toCharArray();
            byte[]              salt = new byte[] {
                                  0x7d, 0x60, 0x43, 0x5f,
                                  0x02, (byte)0xe9, (byte)0xe0, (byte)0xae };
            int                 iterationCount = 2048;
            PBEKeySpec          pbeSpec = new PBEKeySpec(
                                                password, salt, iterationCount);
            SecretKeyFactory    keyFact =
                      SecretKeyFactory.getInstance("PBEWithSHAAnd3KeyTripleDES", "BC");

            Cipher cDec = Cipher.getInstance("PBEWithSHAAnd3KeyTripleDES", "BC");
            Key    sKey = keyFact.generateSecret(pbeSpec);

            cDec.init(Cipher.DECRYPT_MODE, sKey);

            System.out.println("cipher : " + Utils.toHex(out));
            System.out.println("gen key: " + Utils.toHex(sKey.getEncoded()));
            System.out.println("gen iv : " + Utils.toHex(cDec.getIV()));
            System.out.println("plain  : " + Utils.toHex(cDec.doFinal(out)));
        }
    }
```

If you run the example, you should see the following output:

```
cipher : a7b955896f750665ba71eb50ac3071d9832a8b02760c600bf619a75a0697c87c
gen key: 732f2d33c801732b7206756cbd44f9c1c103ddd97c7cbe8e
gen iv : b07bf522c8d608b8
plain  : 000102030405060708090a0b0c0d0e0f0001020304050607
```

Note that this time `gen key` is now the actual DESede key.

How It Works

This is possible because the `SecretKeyFactory` now has all the information required to create a proper key, rather than being able to do only some basic preprocessing and leaving the final pass to the `Cipher` object to perform using the information in the `PBEParameterSpec`. Being able to do this is an option the Bouncy Castle provider takes full advantage of, as this can be useful when you are carrying out encryption using a variety of applications. While all of the applications may support encryption — say, in the form of Three-Key Triple-DES — they may not all support PBE. In such a situation you may need to know what key was actually produced by the PBE key generation process as well as what IV was used at the start of the encryption process. As you can see here, with the newer method it becomes possible to get both the real encoded key and, after calling `Cipher.init()`, the required IV as well — an enormous improvement.

Key Wrapping

As you have probably already noticed from the JavaDoc for the `Cipher` class, in addition to `Cipher.ENCRYPT_MODE` and `Cipher.DECRYPT_MODE`, the `Cipher` class also has two other modes, `Cipher.WRAP_MODE` and `Cipher.UNWRAP_MODE`.

The wrap modes are provided for the purpose of allowing providers to provide facilities for "key wrapping," or the encryption of the encoded form of the keys. There are two reasons for doing this. The first is simple convenience — you do not have to extract the key's data; to wrap it, you just call `Cipher.wrap()` and the key is extracted for you and returned as an encrypted byte array. The second reason is that some providers will store the actual key material on hardware devices where it is safe from prying eyes; the wrapping mechanism provides a means of getting the key material out of the device without exposing the raw material unencrypted. The alternative would be to force the provider to return the key material using `Key.getEncoded()`, not really acceptable if you have gone to the expense of investing in hardware adapters to protect your keys.

The `Cipher.unwrap()` method is provided to reconstruct the key from the encrypted key material. It is slightly more complicated than the `Cipher.wrap()` method in that it expects the algorithm name and the type of the key. The key types make up the other constants provided in the `Cipher` class. They are: `Cipher.PUBLIC_KEY`, `Cipher.PRIVATE_KEY`, and `Cipher.SECRET_KEY`. The algorithm name should be a string that is meaningful to the provider you are creating the key for.

Try It Out — Symmetric Key Wrapping

You will see an example of the use of `Cipher.PUBLIC_KEY` and `Cipher.PRIVATE_KEY` in Chapter 4. In the meanwhile, the following example shows basic key wrapping as applied to symmetric keys:

```
package chapter2;

import java.security.Key;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;

public class SimpleWrapExample
{
```

```
    public static void main(String[] args) throws Exception
    {
        // create a key to wrap
        KeyGenerator generator = KeyGenerator.getInstance("AES", "BC");
        generator.init(128);

        Key    keyToBeWrapped = generator.generateKey();

        System.out.println("input    : " +
                                    Utils.toHex(keyToBeWrapped.getEncoded()));

        // create a wrapper and do the wrapping

        Cipher cipher = Cipher.getInstance("AESWrap", "BC");

        KeyGenerator keyGen = KeyGenerator.getInstance("AES", "BC");
        keyGen.init(256);

        Key wrapKey = keyGen.generateKey();

        cipher.init(Cipher.WRAP_MODE, wrapKey);

        byte[] wrappedKey = cipher.wrap(keyToBeWrapped);

        System.out.println("wrapped  : " + Utils.toHex(wrappedKey));

        // unwrap the wrapped key

        cipher.init(Cipher.UNWRAP_MODE, wrapKey);

        Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);

        System.out.println("unwrapped: " + Utils.toHex(key.getEncoded()));
    }
}
```

Run this and you should see that after the key to be wrapped is printed, an encrypted wrapped key will be printed that is longer than the original key, and then, on the last line, you will see that after unwrapping the original key has been recovered.

How It Works

The easiest way to see how this process works is to consider how you would do it using encryption and decryption. For example, you could replace

```
        Cipher cipher = Cipher.getInstance("AESWrap", "BC");
```

with

```
        Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding", "BC");
```

You could then replace the call to `cipher.init()` and the `cipher.wrap()` methods with

```
            cipher.init(Cipher.ENCRYPT_MODE, wrapKey);

            byte[] wrappedKey = cipher.doFinal(keyToBeWrapped.getEncoded());
```

and rather than use `cipher.unwrap()` to the recover the key, replace it and the second `cipher.init()` with

```
            cipher.init(Cipher.DECRYPT_MODE, wrapKey);

            Key key = new SecretKeySpec(cipher.doFinal(wrappedKey), "AES");
```

and you would achieve almost the same effect.

In essence, the wrapping mechanism calls `Key.getEncoded()` on the symmetric key under the covers and encrypts what `getEncoded()` returns. When the key is unwrapped, the key is reassembled using the extra information passed into the `unwrap()` method and the encrypted bytes. The `SecretKeySpec` class provides you with a general way of assembling a symmetric key, although the provider may use some other mechanism internally if it suits it.

Using the `NoPadding` works, in this case, as the key being wrapped is a multiple of the block size. You will notice if you compare the output from `SimpleWrapExample` with the output from the version modified to use `"AES/ECB/NoPadding"` that the wrapped key in `SimpleWrapExample` appears to have some padding added to it — the wrapped text is longer than the input text. The reason is that the purpose-built key-wrapping mechanism includes an integrity check that is used to ensure that the key probably decrypted properly. A symmetric key is just a string of random bytes and one string of random bytes looks very much like another; so, in general, if you have a purpose built wrapping mechanism available to you, it is better to use it than try to roll your own.

> **Use purpose-built key-wrapping mechanisms where you can for wrapping keys.**

The last thing to note in the example, modified or otherwise, is that the key doing the wrapping is a larger bit size than the key being wrapped. If it were the other way around, it would be easier to guess the wrapping key than guess the key being wrapped. Put another way, if you were to wrap a 256-bit AES key using a 40-bit ARC4 key, you only have 40 bits of security, not 256, protecting the data encrypted with the AES key.

> **Keys used for wrapping should always be at least as secure, if not more so, than the key being protected.**

Doing Cipher-Based I/O

The JCE contains two classes for doing I/O involving ciphers: `javax.crypto.CipherInputStream` and `javax.crypto.CipherOutputStream`. These classes are not only useful but very easy to use as well. You can use them anywhere you would use an `InputStream` or an `OutputStream`.

The stream classes are a case where the usual factory pattern seen elsewhere in the JCE is not used. Instances of both `CipherInputStream` and `CipherOutputStream` are created using constructors that take an `InputStream`, or `OutputStream`, to wrap, and a `Cipher` object to do the processing.

Try It Out Using Cipher-Based I/O

Look at the following example, which uses `CipherInputStream` and `CipherOutputStream`.

```
package chapter2;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * Basic IO example with CTR using AES
 */
public class SimpleIOExample
{
    public static void main(
        String[]    args)
        throws Exception
    {
        byte[]          input = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };
        byte[]          keyBytes = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
                            0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };
        byte[]          ivBytes = new byte[] {
                            0x00, 0x01, 0x02, 0x03, 0x00, 0x01, 0x02, 0x03,
                            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 };

        SecretKeySpec   key = new SecretKeySpec(keyBytes, "AES");
        IvParameterSpec ivSpec = new IvParameterSpec(ivBytes);
        Cipher          cipher = Cipher.getInstance("AES/CTR/NoPadding", "BC");


        System.out.println("input : " + Utils.toHex(input));

        // encryption pass

        cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);

        ByteArrayInputStream    bIn = new ByteArrayInputStream(input);
        CipherInputStream       cIn = new CipherInputStream(bIn, cipher);
        ByteArrayOutputStream   bOut = new ByteArrayOutputStream();
```

```
            int    ch;
            while ((ch = cIn.read()) >= 0)
            {
                bOut.write(ch);
            }

            byte[] cipherText = bOut.toByteArray();

            System.out.println("cipher: " + Utils.toHex(cipherText));

            // decryption pass

            cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);

            bOut = new ByteArrayOutputStream();

            CipherOutputStream      cOut = new CipherOutputStream(bOut, cipher);

            cOut.write(cipherText);

            cOut.close();

            System.out.println("plain : " + Utils.toHex(bOut.toByteArray()));
        }
    }
```

Running the example produces the following:

```
input : 000102030405060708090a0b0c0d0e0f00010203040506
cipher: bbfe17383cc002047c11be5dfc524e4ead5f2a887d197b
plain : 000102030405060708090a0b0c0d0e0f00010203040506
```

How It Works

The example demonstrates the flexibility of the stream model by reading from a stream of plaintext that is encrypted as it is read, and then writing a stream of ciphertext through a suitably configured output stream that will decrypt the ciphertext as it passes it through. `CipherInputStream` and `CipherOutputStream` simply wrap the streams passed to their constructors and then filter anything read, or written, to them through the `Cipher` object passed to their constructor as appropriate.

There is really only one important point to remember with cipher streams. If `close` on the stream is not called, `Cipher.doFinal()` will not be called on the underlying cipher either. For example if the line

```
            cOut.close();
```

is removed from the example, the output would probably look more like the following:

```
input : 000102030405060708090a0b0c0d0e0f00010203040506
cipher: bbfe17383cc002047c11be5dfc524e4ead5f2a887d197b
plain : 000102030405060708090a0b0c0d0e0f
```

The reason for the missing data in the final output is that, because `Cipher.doFinal()` is not called, the `Cipher` object `cipher` never flushes the bytes it is holding on to as it tries to assemble a block.

> **Forgetting to call `CipherOutputStream.close()` or `Cipher.doFinal()` is a very common error. If you find your messages truncated with a block or so of data missing from the end, make sure `close()` or `doFinal()` have been called.**

Summary

You have looked at JCE support for symmetric key encryption and the mechanism by which `Cipher` objects are created so that they will function in particular modes and use specified padding mechanisms if required.

Over the course of the chapter you learned the following:

- Some modes such as CBC mode and CTR mode also require an initialization vector (IV) to be set.
- IVs can be generated by a `Cipher` object automatically or passed in using an `IvParameterSpec` or a suitable `AlgorithmParameters` object. Where the IV is generated automatically by the `Cipher` object, it can be recovered using `Cipher.getIV()` or `Cipher.getAlgorithmParameters()`.
- A symmetric key can be created from raw bytes, using the `SecretKeySpec` class, or generated randomly using the `KeyGenerator` class.
- Keys can also be created from passwords using password-based encryption (PBE).
- Key-wrapping mechanisms can be used for the safe transport of symmetric keys by encrypting them using other symmetric keys.
- How to integrate `Cipher` objects with Java I/O streams by using the `CipherInputStream` class and the `CipherOutputStream` class.

One problem that symmetric key encryption does not address is making sure that the ciphertext has not been tampered with. Providing encryption and decryption mechanisms only solve part of the problem. You will start looking at mechanisms for dealing with this in the next chapter.

Exercises

1. A colleague has written a program for decrypting a padded byte stream that was created by encrypting with a block cipher. For some reason the program the colleague has written is appending one or more zero bytes to the data created by decrypting the stream. What is the most likely reason for the extra bytes? How would you fix the program?
2. You have written a program that is decrypting a block cipher encrypted stream created using CBC mode. For the most part, the data appears to be encrypting correctly, but the first block of the decrypted data is always wrong. What is the most likely cause of this?

3. If you have a `Cipher` object initialized in encryption mode that uses an IV, what are the two ways you can retrieve the IV's value?
4. If you have a `Cipher` object that is using PBE, how would you retrieve the parameters passed to the key generation function, other than the password?
5. What is the most likely problem if data written through a `CipherOutputStream` appears to be truncated?