1

The JCA and the JCE

The basic functionality for using cryptographic techniques in Java is provided by the Java Cryptography Architecture (JCA) and its sibling, the Java Cryptography Extension (JCE).

In this chapter, you will look at how the architecture works, as well as why the architecture is the way it is. You will look at the policy files associated with the JCA and JCE, how signing of cryptographic services providers works, and how the JCA and JCE decide which provider to use if an algorithm you have requested is implemented by more than one. Finally, you will see how to install a provider either statically in the Java runtime or dynamically when an application starts up.

In this chapter, you will learn

- ❑ How the JCA and JCE are architected
- ❑ What patterns apply to their use as an API
- ❑ What a cryptographic services provider is
- ❑ How to add a services provider to your Java runtime, either dynamically or by static configuration changes
- ❑ How to confirm a provider has been installed and what its capabilities are
- ❑ What the common issues are if there are problems

Basic Architecture

The first thing you will notice about the JCA and the JCE is that there is little reference to actual algorithm implementation in the classes and interfaces that make them up. Instead, the JCA and, subsequently, the JCE is architected to provide an abstraction layer for application developers, and the objects that provide the implementations of the algorithms you wish to use are created using factory classes.

This architecture is what is referred to as *provider-based architecture*; in this case, it means that the JCE and JCA provide a set of classes and interfaces that an application developer writes to, together with factories that enable the creation of the objects that conform to the interfaces and classes. The objects that ultimately give the functionality that the application developer is using are provided by an underlying implementation through a *factory pattern* and are not directly visible to the developer. In the JCA and JCE, the collections of classes that provide these implementation objects are, not surprisingly, called *providers*, and the JCA and JCE have some simple mechanisms for allowing people to add providers and to choose specific providers.

A quick look at Figure 1-1 shows how the various parts work together. Application code is written that calls the appropriate JCE/JCA API classes; these in turn invoke the classes in a provider that provides implementations for the JCE/JCA service provider interface (SPI) classes. The classes then invoke the internal code in the provider to provide the actual functionality requested.

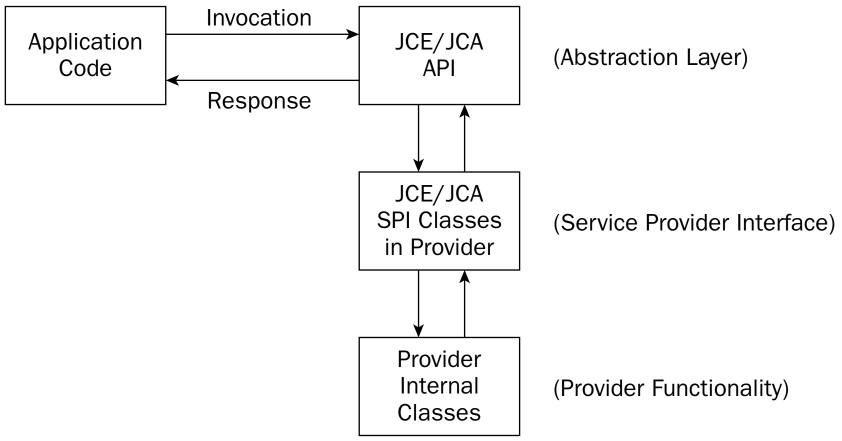

Figure 1-1

This might seem like a roundabout way of doing things at first, but it does make sense after a bit of thought as to why. It helps you see why it is important to understand two major items in the history of the APIs.

The first is that at the time the JCE and JCA were originally developed, export controls on encryption technology, at least in the United States, were a lot stricter than they are now. This is the main reason why the original JCA concerned itself only with authentication, which is generally exempt as long as the underlying algorithms cannot be used for encryption. It is also the reason that the JCE ended up as an extension, rather than as was originally intended, part of the regular security APIs. Export control extends not only to underlying implementations but also to what is referred to as “crypto in the hole” or, more clearly, frameworks that can be used for encryption if some magic component — in this case a cipher implementation — is added later. In this case, as the JCE was a framework for doing encryption, it was covered by “crypto in the hole.” Hence, it was deemed to be covered by cryptography export laws, to the point where the original specification was actually presented as a white paper at a conference in May 1997, academic presentation being the only legal means of publishing it.

The next issue was that algorithms like RSA were still under patent, so it was not possible to include implementations of them directly; however, the API had to be able to allow people to write code that dealt with things like RSA signatures and ciphers as well as the padding and hashing mechanisms that accompany them. In the light of this, having the facility for pluggable providers accessed via a factory pattern not only makes sense but is a necessity.

A consequence of the use of the factory pattern is that it makes it very simple for people to plug and play with providers, which can be very useful. For example, you may be planning to deploy an application using a hardware cryptography device, be it external or a bus adapter. In this case the device will have a thin JCA/JCE provider built on top of a JNI interface. On the other hand, you will probably find it a lot easier and cheaper to allow your developers to use a software-only provider when developing the application. Another consequence is that, at least from an application developer's point of view, it does make the API easier to learn: All ciphers look the same, all cryptographic hashes look the same, and the only change that needs to be made is the string passed to the factory to select the implementation. Further, new implementations of algorithms can be added without increasing the apparent complexity of the API. As an example of the kind of "complexity creep" I am talking about here, it is worth considering that in the Bouncy Castle Lightweight API there are 23 engine classes in the `org.bouncycastle.crypto.engines` package, all of which are encapsulated by the single `Cipher` class in the JCE.

Now you are probably wondering what changed that allowed the JCE to go from being something that was not available in other than clean room implementations outside of the United States to something that is now being included with the JDK. Well, cryptography export controls relaxed a bit and at the same time changes were made to the JCE being shipped by Sun. This allowed the JCE to meet the relaxed controls. Sun introduced the use of policy files, which allowed the use of algorithms and key sizes to be restricted and also introduced the idea of signed providers. The changes were signed off by the National Security Agency (NSA) and the U.S. Department of Commerce, and here we are today: JCE 1.2.2, which is for JDK 1.2 and JDK 1.3, ships with unrestricted policy files and works with any signed provider, and every JDK that has shipped since JDK 1.4 ships with the JCE included and restricted policy files that allow access to certain key sizes. The JDK 1.4 and later versions of the JCE will work with any correctly signed provider and have unrestricted policy files available for them. Considering that there are a number of commercial and open source providers that are now signed, when you consider where we were five years ago, this is quite a good result!

Note I have said that the JDK since 1.4 ships with restricted policy files. The reason for this has nothing to do with U.S. cryptography export laws; it is because the JDK ships only with the key sizes that all countries it can be imported into find acceptable. If you need to run the JCE without restrictions, you need to download the unrestricted policy files separately, if it is legal in the country where you are.

This brings us to the two lessons that can be learned from this history. As mentioned, export controls in the United States have relaxed, but it does not mean they do not exist, nor does it mean that if you are not in the United States, your local government does not have laws in with regard to the development of encryption technology. If you want to play in this space, possibly developing products that may be exported overseas, or in some cases even used within your nation's borders, make sure you understand what the legal situation is first. Encryption technology, including software and hardware, is still widely regarded as munitions, and consequently the penalties for ignoring the law on the export and use of encryption technology are very similar to those handed out for arms smuggling. In addition, some algorithms, such as the symmetric cipher IDEA, are still patented in some countries. If you use an algorithm in a country where it is still patented, you are obliged to pay royalties regardless of whether you paid for the software.

So, enough about history, the next things I need to cover in dealing with the JCA and the JCE are the issues of provider signing and the workings of the JCE policy files.

Provider Signing

Provider signing is enforced using root certificates that are embedded in the Java runtime. If you want to write a provider for use with the Java runtime, you need to create a private key and get an associated certificate created, which is signed by a certificate chain that starts with one of the root certificates embedded in the Java runtime.

I will not go into any real detail on provider signing here, as it is only relevant if you need to create your own provider jars. In brief, getting a signing certificate to validate a provider requires getting a certificate from Sun; you can find the details on how to do this in the document entitled "How to Implement a Provider for the Java Cryptography Extension," which is distributed with the document set for the JDK. (This is normally in a file called `docs/guide/security/HowToImplAProvider.html` under the Java documentation tree.) Apart from this fact, the other item you need to be aware of is that if you need to take advantage of the way policy files are used by the Java runtime, you can do so by creating your own application jar and incorporating a policy file. You might want to use an existing code base and create a special provider as part of the application you ship. In any case, the details can be found in the document "Java Cryptography Extension (JCE) Reference Guide" (normally found in `docs/guide/security/jce/JCERefGuide.html` under the Java documentation tree). This can be important if you need to develop a product that must conform to either corporate or government restrictions and enforce algorithm restrictions, limited key sizes, mechanisms such as key escrow, or some combination of all three.

Jurisdiction Policy Files

The normal JDK download ships with a set of policy files that places certain restrictions on the key sizes that can be used. Key sizes are limited in general to 128 bits (except for the symmetric cipher Triple-DES) and RSA key generation is limited to 2,048 bits. The easiest way to deal with this restriction if it need not apply to you is to download the unrestricted policy files.

Installing the Unrestricted Policy Files

The following information applies only if you are dealing with JDK 1.4 or later; if you are using JDK 1.2 or JDK 1.3 and using the JCE 1.2.2, the unrestricted policy files come preinstalled in the JCE 1.2.2 distribution.

You can find the unrestricted policy files on the same page as the JCE/JDK downloads are found. Normally it will be a discrete link at the bottom of the download page entitled something like

"Unlimited Strength Jurisdiction Policy Files." The download is a ZIP file, and providing it is legal for you to do so; you should download the ZIP file and install the two JAR files it contains according to the instructions in the README file contained in the ZIP file.

If you are installing these on a Linux, or some other Unix variant, you need to make sure you install the policy files in the Java runtime you are using, and you will probably need root access or the assistance of a root user to do so, unless you have installed a personal runtime of your own.

If you are installing on Windows, you also need to be extra careful about installing the policy files, as the default install for the JDK installs a runtime for the JDK and a separate JRE, which will normally appear under `C:\Program Files\Java`.

Try It Out Testing that the Policy Files Are Installed

Once you have installed the policy files, it is a good idea to make sure they really are installed where you think they are. Try typing and running the following test program. It will not only tell you if the policy files are installed but give you an introduction to the use of the `Cipher` class and the simple creation of `SecretKey` objects. There is also an electronic version available for download on the book's Web site.

```
package chapter1;

import javax.crypto.*;
import javax.crypto.spec.*;

public class SimplePolicyTest
{
    public static void main(String[]  args) throws Exception
    {
        byte[]     data = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };

        // create a 64 bit secret key from raw bytes
        SecretKey  key64 = new SecretKeySpec(
                    new byte[] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 },
                    "Blowfish");

        // create a cipher and attempt to encrypt the data block with our key
        Cipher     c = Cipher.getInstance("Blowfish/ECB/NoPadding");

        c.init(Cipher.ENCRYPT_MODE, key64);
        c.doFinal(data);
        System.out.println("64 bit test: passed");

        // create a 192 bit secret key from raw bytes
        SecretKey  key192 = new SecretKeySpec(
                     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 },
                     "Blowfish");

        // now try encrypting with the larger key
        c.init(Cipher.ENCRYPT_MODE, key192);
        c.doFinal(data);
```

```
            System.out.println("192 bit test: passed");


            System.out.println("Tests completed");
        }
    }
```

If the files are installed, your class path is correct, and you have compiled the test program, it should be a matter of executing

```
java chapter1.SimplePolicyTest
```

If all is well you will see

```
64 bit test: passed
192 bit test: passed
Tests completed
```

On the other hand if it does not work the most likely thing you will see is

```
64 bit test: passed
Exception in thread "main" java.lang.SecurityException:
        Unsupported keysize or algorithm parameters
        at javax.crypto.Cipher.init(...)
        at chapter1.SimplePolicyTest.main(SimplePolicyTest.java:38)
```

If the preceding happens, it means the policy files are not installed correctly. Check the README file that came with the unrestricted policy file distribution and make sure it has been followed.

Blowfish has been chosen as the cipher to use above because it allows for large keys and is present in all versions of the provider that comes by default with the Sun JCE, as well as the clean-room JCE implementations that are available. I will deal with what algorithms are useful to use in production code later.

How It Works

Whenever an attempt is made to utilize any of the functionality in a JCE provider, the runtime code supporting the API checks the policy restrictions associated with that piece of functionality. In this example, one of two things can happen:

- It will work, in which case the Java runtime has determined from the JCE policy files that the key size you requested to use with the `Cipher` class is permissible.
- It will fail, in which case the Java runtime will have determined from the JCE policy files that the key size you requested to use is not permissible.

You probably noticed that the exception also included the term *algorithm parameters* in its message. This will make sense as you go further. You will see that key size is not the only way of expressing the arguments that are used to initialize objects, like `Cipher`, that provide the functionality available to you in the JCE.

Troubleshooting Other Issues

You may also run into some other problems while compiling or running the test. The most likely ones are presented here:

- **Test does not compile, error message is** `package javax.crypto does not exist`**.** If this happens, the JCE is not installed. Make sure there is a JAR file containing the `javax.crypto` package tree in either the `jre/lib/ext` (`jre\lib\ext`) directory or the `jre/lib` (`jre\lib`) directory under your install of Java. If you see this and you are running JDK 1.4 or later, there is a problem with your Java installation, and you should reinstall it. If you are running JDK 1.2 or JDK 1.3 and you do not have the JCE, you can download it from `http://java.sun.com`— the release you will be looking for in this case is the JCE 1.2.2.
- **Test compiles; however, the runtime exception `java.security.NoSuchAlgorithm Exception: Algorithm Blowfish not available` occurs.** In this case the problem is with the provider. Check that the `java.security` files in your installation have not been tampered with and the SunJCE provider is present. Once again, if this happens and you are running Java 1.4 or later, there is a problem with your installation. If you are running JDK 1.2 or JDK 1.3, check that the install instructions for the JCE have been followed properly.

If you see any other problems, the most likely causes are as follows:

- **If you are running JDK 1.4 or later.** These come with the JCE preinstalled. There will be a problem with your installation and it would probably be simpler to reinstall it.
- **If you are running JDK 1.2 or JDK 1.3.** For these the JCE would have had to have been installed by hand. Check that the installation instructions have been followed correctly.

How Do You Know the Policy Files Really Behave as Sun Says They Do?

When you look at the history behind the JCE, it is easy to see why some people would be inclined to feel it is all a conspiracy by Sun, the U.S. government, the UN, the Illuminati, the Greys, or some other “organization” and that anything running under the JCE should not be trusted, as you cannot see the code that deals with key strength and various other control mechanisms that the policy files allow you to turn on or off. If this is a concern for you, remember, just as you would test a hardware cryptography adapter by comparing its outputs for given inputs against the outputs produced by an alternative implementation for the same set of outputs, you can also treat the JCE and the underlying provider as a black box and perform the same tests. As it happens the Bouncy Castle provider was first developed as a lightweight library for MIDP, so if you are feeling really enthused, you can start by verifying that the Bouncy Castle provider and the JCE is producing the same output as its lightweight equivalent. It is true that in matters like these you should not take what people tell you at face value, but always remember you can test and investigate the truth of other people’s claims yourself.

Installing the Bouncy Castle Provider

There are two ways to install a provider for the JCE and JCA. You can do it dynamically at runtime, or you can configure the JRE to preload the provider for you so that it is “naturally” available in the environment. Which one is best for you will vary according to circumstance, but in general, you will experience fewer issues if you install providers by configuring the Java runtime.

Note that while this book is largely written for use with the Bouncy Castle provider, the installation instructions that follow should work for any appropriately signed JCE/JCA provider.

Installing by Configuring the Java Runtime

This is by far the simplest and most convenient mechanism for dealing with a provider. On the downside, it also involves changing the configuration of the Java runtime.

There are two steps involved in installing by configuring the Java runtime. First you need to install the JAR file containing the provider, and then you need to enable the provider so the Java runtime can find it.

Install the JAR File Containing the Provider

Under your Java installation you should find the directory `jre/lib/ext`, or if you are using Windows, `jre\lib\ext`. The purpose of this directory is to provide a home for standard extensions to the runtime, which are not normally distributed with the default setup. This is where the provider JAR should go.

In the case of the Bouncy Castle provider you can find the JAR file you need on the page `http://www.bouncycastle.org/latest_releases.html`. The naming convention used for provider JARs in Bouncy Castle is `bcprov-JdkVersion-Version.jar`. So for example, if you were after the JDK 1.4 provider JAR and the release you were looking for was version 1.29, the name of the JAR file you want to download would be `bcprov-jdk14-129.jar`.

Download the JAR file and copy it to the `jre/lib/ext` directory of your Java install. Remember, if you are using Windows, you will probably have two installations, one for the full JDK and one that just contains the JRE, which will normally be under your `Program Files` directory tree. If you have an install of the JRE, you will need to make sure the provider is present under its `lib/ext` directory.

Enable the Provider by Adding It to the java.security File

If you have already been through the installation of the unrestricted policy files, you will probably have noticed that in addition to `jre/lib/ext` under every Java install, there is also a `jre/lib/security`. One of the files contained in `jre/lib/security` specifies, amongst other things, what providers are enabled for the JCE and JCA, as well as their *precedence*. We will look at what precedence means in the next section but for now it is enough to say if you open the file for editing and scroll down a bit, you will see a group of lines like the following:

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.sun.rsajca.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
```

The list may have more, or fewer, providers largely depending on which version of Java you have installed. Add the following line to the end of the list:

```
security.provider.N=org.bouncycastle.jce.provider.BouncyCastleProvider
```

`N` needs to be the next number in the sequence. So, for the previous example list, you would add the line:

```
security.provider.5=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Save the file after making the change, again being making sure you have changed the `java.security` files in all the installs of Java you are using (remember, you may have two installs under Windows, in which case the second one will be under `lib/security` of your JRE install), and you are finished.

It might be tempting to put the Bouncy Castle provider first and move all the other ones down one. Do not do this; some Java system software rely on the Sun providers being the first ones in the list, and things will stop working if they aren't.

Try It Out Verifying Bouncy Castle Provider Installation

If you have added the Bouncy Castle provider to the `java.security` file and correctly installed the provider JAR file in `jre/lib/ext`, compiling and running the following program, `SimpleProviderTest`, will indicate everything is correct by printing

```
BC provider is installed
```

If the program prints the message `BC provider not installed` instead, first check that the provider has been added to the `java.security` file in `jre/lib/security`, and then check that the provider JAR has been installed in the `jre/lib/ext` directory. You also need to be sure that only one version of the provider has been installed — for example, if you are dealing with JDK 1.4, having both `bcprov-jdk14-128.jar` and `bcprov-jdk14-129.jar` in `jre/lib/ext` at the same time will cause confusion and the provider will not work as expected.

```
package chapter1;

import java.security.Security;

/**
 * Basic class to confirm the Bouncy Castle provider is
 * installed.
 */
public class SimpleProviderTest
{
    public static void main(String[] args)
    {
        String providerName = "BC";

        if (Security.getProvider(providerName) == null)
        {
            System.out.println(providerName + " provider not installed");
        }
        else
        {
            System.out.println(providerName + " is installed.");
        }
    }
}
```

How It Works

If you look back to the changes made to the `java.security` configuration file, you will see that the following line was added:

```
security.provider.5=org.bouncycastle.jce.provider.BouncyCastleProvider
```

The `Provider` class has a `getName()` method on it that returns a simple name for the provider. It is this simple name that the JCE/JCA classes in the Java runtime are trying to match when a request to create an object is made using the `getInstance()` factory pattern that includes a provider name. In the case of the `BouncyCastleProvider` class, its simple name is just `BC`.

You can also use this simple name to get a copy of the provider via the static `Security.getProvider()` method. Because it is provider implemented by the `BouncyCastleProvider` class that you are interested in, you just pass its simple name (`BC`) to `Security.getProvider()` to retrieve it. Of course, if there is no match for the simple name `BC` and a `null` is returned, it means the provider has not been installed correctly.

As a useful aside, the `Security` class also allows you to retrieve an array of all the providers available using the static `Security.getProviders()` method.

Installing During Execution

A provider can also be added during the execution of your program. This is done via the class `java.security.Security` using the `addProvider()` method. In the case where you wanted to add the Bouncy Castle provider at runtime, you could add the following imports:

```
import java.security.Security;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
```

Then insert the line:

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

This line must be added before the provider is referred to in your program. For example, if you were modifying the previous test program, you would obviously need to use the `Security.addProvider()` method before attempting to use `Security.getProvider()`.

The examples in the book and on the book Web site assume you have installed the Bouncy Castle provider by configuring the JRE. If you have not, you will need to modify the examples according to the previous instructions for them to work correctly.

How Provider Precedence Works

You saw earlier that the `java.security` configuration file had precedence numbers associated with each provider installed. Now you will look at how precedence is used. When you instantiate a JCA/JCE object that is created using the `getInstance()` factory pattern, you can either specify the provider you wish to use or leave it up to the Java runtime to choose the provider for you. So, if you wish to specify you want to use the Bouncy Castle provider to create the object, you mightsay, in the case of a `Cipher` object:

```
Cipher.getInstance("Blowfish/ECB/NoPadding", "BC");
```

Or if you do not care which provider is used to create the object you want, you might say

```
Cipher.getInstance("Blowfish/ECB/NoPadding");
```

In this case the Java runtime will select the first provider it can find that can satisfy the request, according to the list of providers in the `java.security` file and the provider's precedence as given there. Providers with a lower preference number take precedence over those with a higher one.

Try It Out Precedence Demonstration

To demonstrate how precedence rules apply, try running the following program:

```
package chapter1;

import javax.crypto.Cipher;

/**
 * Basic demonstration of precedence in action.
 */
public class PrecedenceTest
{
    public static void main(String[] args) throws Exception
    {
        Cipher        cipher = Cipher.getInstance("Blowfish/ECB/NoPadding");

        System.out.println(cipher.getProvider());

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

        System.out.println(cipher.getProvider());
    }
}
```

The output will probably look something like:

```
SunJCE version 1.42
BC version 1.29
```

How It Works

Although the version numbers may be slightly different, the principle is the same: If you do not specify a particular provider, the provider with the lowest preference order will be used. The SunJCE provider has a higher precedence than the BC provider, so if you do not specify a provider for the Blowfish cipher, you will always get the SunJCEs version.

In general, if you are deploying an application, it is well worth your while to specify the provider in addition to the full algorithm name for the JCA/JCE objects you want to use. Relying on precedence rules can get you into trouble if there are incompatibilities that you are not aware of between providers that you might be mixing or incompatibilities between your code and other providers.

Examining the Capabilities of a Provider

Providers make their capabilities available to the JCA and JCE using a property table that is publicly available, so it is a fairly simple process to write a program that will tell you what a given provider provides support for.

Try It Out Listing Provider Capabilities

The following program dumps out the base algorithm names and the factory classes so they can be used with the BC provider. As you can see, it just iterates through the entries in the property table the provider contains.

```
package chapter1;

import java.security.Provider;
import java.security.Security;
import java.util.Iterator;

/**
 * List the available capabilities for ciphers, key agreement, macs, message
 * digests, signatures and other objects in the BC provider.
 */
public class ListBCCapabilities
{
    public static void main(
        String[]    args)
    {
        Provider      provider = Security.getProvider("BC");

        Iterator  it = provider.keySet().iterator();

        while (it.hasNext())
        {
            String     entry = (String)it.next();

            // this indicates the entry actually refers to another entry

            if (entry.startsWith("Alg.Alias."))
            {
                entry = entry.substring("Alg.Alias.".length());
            }

            String  factoryClass = entry.substring(0, entry.indexOf('.'));
            String  name = entry.substring(factoryClass.length() + 1);

            System.out.println(factoryClass + ": " + name);
        }
    }
}
```

How It Works

As you saw before, `Security.getProvider()` allows you to retrieve a provider based on its simple name. In the case of a `Provider` implementation, it is the key set for the provider's property table that contains the types associated with each entry, and that determines whether that entry is simply an alias for another one. The result is that you can get a fairly clear idea of what a provider supports from the strings contained in its property table's key set.

When you run the program, you will probably notice there is quite a bit of output. I will not go into detail about exactly what the output means here, other than to say it will be covered in later chapters. Basically, however, if you see

```
Cipher: AES
```

the line means that a `Cipher.getInstance()` call like

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

or some variant, should produce an object, likewise for MessageDigest, Mac, SecretKeyFactory, and so on.

You can find out more information about the provider capabilities tables, such as how the aliasing works in the "How to Implement a Provider for the Java Cryptography Extension" document referred to earlier in the chapter.

Summary

This chapter has been aimed at making sure you are properly set up to understand and practice what is discussed in the rest of the book.

You have had a brief look at how the JCE and JCA APIs evolved the way they have, and a discussion about how to utilize the policy mechanisms that they provide. As mentioned earlier, for the most part configuring policy for JCE and JCA providers will primarily be a matter of just downloading and installing the unrestricted policy files. It is important to remember that you can make use of your own policy files if you need to as well. The details on how to do this ship with the JCE documentation that Sun provides.

In addition to this, in this chapter you learned

- ❑ How to confirm the policy files for the JCE, and that the JCA are configured as you thought they were
- ❑ How to install a provider into your Java runtime and verify this fact
- ❑ How to examine a provider to see what algorithms it contains
- ❑ How the `getInstance()` factory pattern works and how precedence works

In the next chapter, I will begin to expand beyond basic setup to the practicalities of using the JCE, starting with making use of the JCE for doing symmetric key encryption.

Exercises

1. Some colleagues come to you with a problem they are having using a JCE provider they have downloaded. They have installed the provider correctly in the `jre/lib/ext` area and added it to the configuration file in `java.security` but are getting a `java.lang.SecurityException` with the message `Unsupported keysize or algorithm parameters`. What have they forgotten?
2. You are running in a Windows environment. You have downloaded and installed everything into your JDK that is required to support the provider you want to use, but you still find that some Java applications you are using fail to find the provider, even though you are sure you have installed it. Why might this be happening?
3. You are attempting to use an algorithm that is available in a provider you have installed, but you are finding that when you create an object to use it via the `getInstance()` method, the object does not have all the capabilities that the documentation that comes with the provider indicates. Why might this be the case?