CHAPTER 3

Cryptography with Java

Cryptography, or the art, science and mathematics of keeping messages secure, is at the heart of modern computer security. Primitive cryptographic operations such as *one-way hash functions*, also known as *message digests*, and encryption, either with symmetric or asymmetric algorithms, form the basis for higher level mechanisms such as MAC (**Message Authentication Code**), digital signature and certificates. At yet another level, these are merely building blocks for security infrastructure consisting of PKI, secure communication protocols such as SSL and SSH, and products incorporating these technologies.

The study of principles and algorithms behind these cryptographic operations and security protocols is fascinating but of little practical relevance to a Java programmer. A typical Java programmer programs at a much higher level, dealing mostly with the APIs, configuration options, proper handling of cryptographic entities such as *certificates* and *keystores*, and interfacing with other security products to satisfy the application's security needs. At times, there may be decisions to be made with respect to the most appropriate mechanism, algorithms, parameters and other relevant aspects for solving the problem at hand. At other times, the challenge may be to design the application so that it can be deployed under different situations to satisfy different security and performance needs. At yet other times, the primary objective may be simply to achieve the best possible performance, scalability and availability of the application without compromising the level of security by selecting the right security products. Our discussion of cryptography with Java in this and subsequent chapters is structured around this notion of usefulness and practicality to a typical Java programmer.

Two Java APIs, JCA (**Java Cryptography Architecture**) and JCE (**Java Cryptography Extension**) both part of J2SE SDK v1.4, define the general architecture and specific services for cryptographic operations. Among these, JCA was introduced first and specifies the architectural framework for cryptographic support in Java. It also includes Java classes for digital signature,

message digest and other associated services. JCE classes follow the same general structure as JCA classes, and include classes for encryption and decryption, MAC computation and a few others. We discuss the JCA architectural framework and explore various cryptographic services available with JCA and JCE in this chapter. Toward this, we develop simple programs making use of these APIs and look at their source code.

Though we talk about some of the JCA and JCE APIs and present code fragments, the discussion of Java interfaces, classes and methods is anything but exhaustive. Our intent is to get a better view of the overall picture and understand their inter-relations. If you do need the complete information on any specific topic, refer to the J2SE SDK Javadocs and the respective specification documents. Keep in mind that the purpose of this chapter is to make you, a Java and J2EE programmer, feel at home with cryptographic capabilities of Java and not to make you an expert on developing security software.

Example Programs and `crypttool`

As mentioned in the *JSTK (Java Security Tool Kit)* section of the *Preface*, this book is accompanied by a collection of utilities and example programs, termed as JSTK software. This software includes not only the source files of example programs presented throughout this book but also the various utility programs that I wrote in the course of researching and using Java APIs for this book. Refer to *Appendix C* for more information on this software.

Example programs are usually good for illustrating use of specific APIs but are not written for flexible handling of input, output and other user specified parameters. In this book, we come across situations when it would be handy to have a tool that could perform some of the operations illustrated earlier in the text but in a more flexible manner. You will find most operations of this kind available through an appropriate command line tool packaged within JSTK.

Example programs illustrated in this chapter can be found in the directory `%JSTK_HOME%\src\jsbook\ch3\ex1`, where the environment variable `JSTK_HOME` points to the JSTK home directory. The utility program covering most of the operations is **`crypttool`** and can be invoked by command **`"bin\crypttool"`** on a Windows machine and by **`"bin/crypttool.sh"`** on a UNIX or Linux machine, from the JSTK home directory. We talk more about this utility in later in this chapter.

Cryptographic Services and Providers

In Java API terminology, cryptographic services are programming abstractions to carry out or facilitate cryptographic operations. Most often, these services are represented as Java classes with names conveying the intent of the service. For example, digital signature service, represented by `java.security.Signature` class, creates and verifies digital signatures. However, not all services are directly related to cryptographic operations. Take the functionality to

create certificates. This is provided as a certificate factory service through service class `java.security.cert.CertificateFactory`.

An instance of a service is always associated with one of many *algorithms* or *types.* The algorithm determines the specific sequence of steps to be carried out for a specific operation. Similarly, the type determines the format to encode or store information with specific semantics. For example, a `Signature` instance could be associated with algorithm DSA (**Digital Signature Algorithm**) or RSA (named after the initial letters of its three inventors: **Rivest, Shamir, and Adleman**). Similarly, a `CertificateFactory` instance could be associated with certificate type X.509.

While talking about cryptographic services and their algorithms or types, we use the term algorithm for brevity, knowing well that some services will have associated types and not algorithms.

The cryptographic service classes have a distinct structure to facilitate independence from algorithm and implementation. They typically do not have public constructors and the instances are created by invoking a static method `getInstance()` on the service class. The algorithm or type, represented as a string, must be specified as an argument to the `getInstance()` method. For exammple, the following statement creates a `Signature` instance with `"SHA1WithDSA"` algorithm.

```
Signature sign = Siganture.getInstance("SHA1WithDSA");
```

Besides the algorithm, one could also specify the implementation, also known as the provider, while creating an instance of the service. This is illustrated by passing the string `"SUN"`, and identifying a specific provider as an additional parameter.

```
Signature sign = Siganture.getInstance("SHA1WithDSA", "SUN");
```

This structure of the API allows different implementation of the same service, supporting overlapping collections of algorithms, to exist within the same program and be accessible through the same service class. We talk more about this mechanism in the next section.

As noted earlier, certain services require an algorithm whereas others require a type. As we saw, signature service requires an algorithm whereas key store service requires a type. Roughly speaking, a service representing an operation needs an algorithm and a service representing an entity or actor needs a type.

Table 3-1 lists some of the J2SE v1.4 cryptographic services and supported algorithms, with brief descriptions. A comprehensive table can be found in *Appendix B*.

You may find the information in *Table 3-1* a bit overwhelming, but don't be alarmed. We talk more about the various services and supported algorithms later in the chapter. Just keep in mind that cryptographic services have corresponding Java classes with the same names and algorithm identifiers passed as string arguments to method invocations.

The separation of service from algorithm, coupled with the API design where a specific service instance of a particular implementation is obtained by specifying them at runtime, is the key mechanism for algorithm and implementation independence. The visible service API

Table 3-1 Java Cryptographic Services

Cryptographic Service	Algorithms/Types	Brief Description
`SecureRandom`	`SHA1PRNG`	Generates random numbers appropriate for use in cryptography.
`KeyGenerator`	`DES, Triple-DES, Blowfish`	Generates secret keys to be used by other services with the same algorithms.
`KeyPairGenerator`	`DSA, RSA, DH`	Generates a pair of public and private keys to be used by other services with the same algorithms.
`MessageDigest`	`SHA1, MD5`	Computes the digest of a message.
`Mac`	`HmacMD5, HmacSHA1`	Computes the message authentication code of a message.
`Signature`	`SHA1WithDSA, SHA1WithRSA`	Creates and verifies the digital signature of a message
`KeyStore`	`JKS, JCEKS, PKCS12`	Stores keys and certificates.
`CertificateFactory`	`X509`	Creates certificates.
`Cipher`	`DES, Triple-DES, Blowfish`	Encrypts and decrypts messages.
`KeyAgreement`	`DH`	Lets two parties agree on a secret key without exchanging it over an insecure medium.

classes, such as `Signature` and `CertificateFactory`, act only as a mechanism to get to the real implementation class, and hence are also referred to as *engine classes*. We find many examples of such classes later in the chapter and also in subsequent chapters.

Providers

As we noted, Cryptographic Service Providers, or just providers, are implementations of cryptographic services consisting of classes belonging to one or more Java packages. It is possible to have multiple providers installed within a J2SE environment, some even implementing the same service with the same algorithms. A program can either explicitly specify the provider name through an identifier string assigned by the vendor, or implicitly ask for the highest priority provider by not specifying any provider. In the last section, statement `Signature.getInstance("SHA1withDSA")` retrieves the implementation class of Signature implementing algorithm "SHA1withDSA" of the highest priority provider. In contrast, `Signature.getInstance("SHA1withDSA", "SUN")` retrieves the implementation class from the provider with the name `"SUN"`.

A mechanism exists to specify priorities to these providers. We talk more about this mechanism in a subsequent section.

Note that JCA and JCE APIs define only the engine classes, most of them within `java.security`, `javax.crypto` and their various subpackages. The actual implementation of these classes is in various provider classes that come bundled with J2SE v1.4. It is also possible to install additional providers. We learn how to install additional providers in the section *Installing and Configuring a Provider.*

A few points about providers are worth noting. A provider doesn't have to implement all the services defined within JCA or JCE. Also, a provider can implement some services from one API and some from another. Which algorithms are to be supported for a specific service is also left to the provider. As you can see, the bundling of engine classes in separate APIs is quite independent of the packaging of classes within a provider.

Thankfully, there are APIs to access all the available providers, the services supported by them and other associated details. JSTK utility **`crypttool`** has a command to list the providers and related details. But before we get to that, let us understand the mechanism to achieve algorithm and implementation independence by looking at the internal structure of engine classes and their relationship with provider classes.

Algorithm and Implementation Independence

The best way to illustrate this independence is with the help of an example. Take the simple service of creating and verifying a digital signature, `java.security.Signature`. It has a static method `getInstance()` that takes the algorithm name and optionally, the provider name, as arguments and creates a concrete `Signature` object. The client program operates on this object, initializing it for signing by invoking the `initSign()` method or for verification by invoking the `initVerify()` method.

Under the hood, the static method `getInstance()` consults the *singleton class* `java.security.Security` to get the fully qualified name of the class associated with `Signature` service for the specified provider or, if the provider is not specified, the highest priority provider with `Signature` implementation for the specified algorithm. This implementation class must extend the abstract class `java.security.SignatureSpi` and provide the implementation of all the abstract methods. Once such a class name is found, the corresponding object is constructed using Java reflection and passed to the protected constructor of the `Signature` class. The Signature class keeps a reference of the newly created object in its member variable. Subsequent method invocations on `Signature` object operate on the object corresponding to the underlying implementation class.

The relationship of various classes and their runtime behavior is further illustrated in *Figure 3-1*. Class `XYZProvider` extends `java.security.Provider` and registers itself to the singleton class `Security`. This provider supplies the concrete implementation class `XYZSignature`, extending abstract class `SignatureSPI`.

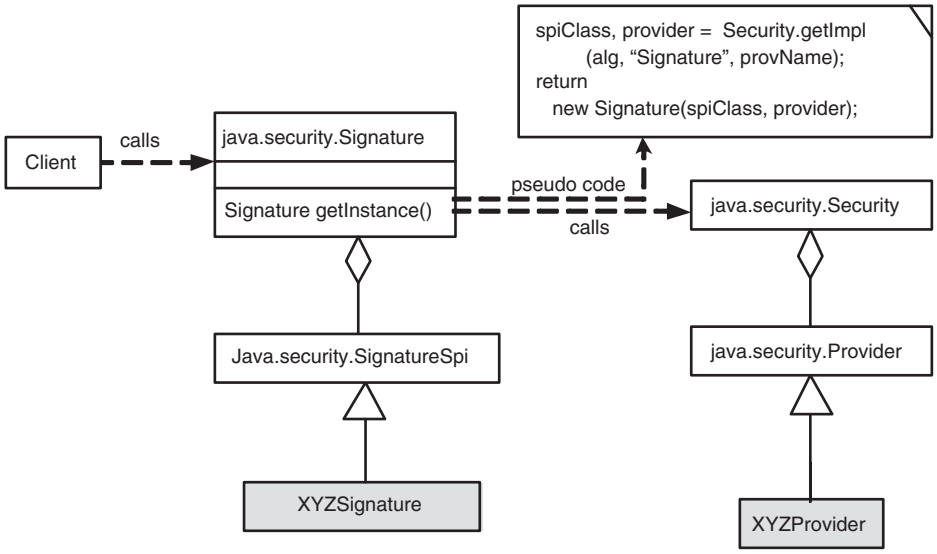

Figure 3-1 Provider Architecture for Signature Class.

Although the preceding discussion and the diagram is for `Signature` service, the same is true for all other cryptographic services. The point to be noted is that even though the client program uses a well-known class, the selection of the actual class implementing the service happens at runtime. This makes adding new providers with new algorithms fairly straightforward and quite transparent to the client program. Well, at least within certain limits. We come across situations when this simple framework breaks down and the client must include code that knows about specific algorithms.

Listing Providers

As we said earlier, it is possible to query a J2SE environment for currently installed providers and the cryptographic services supported by them. This ability comes in handy in writing programs that adjust their behavior based on the capabilities available within an environment and also in troubleshooting.

Java class `Security` keeps track of all the installed providers in the form of `Provider` class instances and can be queried to get this information. A `Provider` object contains entries for each service and information on supported algorithms.

The example program `ListCSPs.java`, available in the examples directory for this chapter `src\jsbook\ch3\ex1`, lists all the installed cryptographic service providers, indicating their name and version.

Listing 3-1 Listing Cryptographic Service Providers

```
// File: src\jsbook\ch3\ex1\ListCSPs.java
import java.security.Security;
import java.security.Provider;

public class ListCSPs {
  public static void main(String[] unused){
    Provider[] providers = Security.getProviders();
    for (int i = 0; i < providers.length; i++){
      String name = providers[i].getName();
      double version = providers[i].getVersion();
      System.out.println("Provider["+i+"]:: " + name + " " +
version);
    }
  }
}
```

Compiling and running this program under J2SE v1.4.x, assuming that you are in the same directory as this file and either the `CLASSPATH` is not set or includes the current directory, produces the following output:

```
C:\ch3\ex1>%JAVA_HOME%\bin\javac ListCSPs.java

C:\ch3\ex1>%JAVA_HOME%\bin\java ListCSPs
Provider[0]:: SUN 1.2
Provider[1]:: SunJSSE 1.41
Provider[2]:: SunRsaSign 1.0
Provider[3]:: SunJCE 1.4
Provider[4]:: SunJGSS 1.0
```

You can infer from the output that J2SE v1.4.1 comes with five bundled providers and their names are: `"SUN"`, `"SunJSSE"`, `"SunRsaSign"`, `"SunJCE"` and `"SunJGSS"`. The same code is executed by utility **`crypttool`** with **`listp`** command, for listing providers.

Information about services implemented by a provider, aliases or different names corresponding to the same service, supported algorithms and other associated properties are stored within the `Provider` object as name value pairs. These name value pairs can be displayed by running the command **`"crypttool listp -props"`**. However, deducing information about each service from this listing is somewhat nontrivial and hence is made available through a separate option **–csinfo**, for cryptographic service information. Let us look at the output of **`"crypttool listp -csinfo"`** command in *Listing 3-2*.

Listing 3-2 Output of "bin\crypttool listp -csinfo" command

```
C:\...\jstk>bin\crypttool listp -csinfo
Provider[0]:: SUN 1.2
Cryptographic Services::
[0] MessageDigest    : SHA1|SHA|SHA-1
                           ImplementedIn = Software
                       MD5
                           ImplementedIn = Software
[1] KeyStore         : JKS
                           ImplementedIn = Software
[2] Signature: SHAwithDSA|DSAWithSHA1|DSA|SHA/DSA|SHA-1/
DSA|SHA1withDSA|
DSS|SHA1/DSA
                           ImplementedIn = Software
                           KeySize = 1024
[3] SecureRandom     : SHA1PRNG
                           ImplementedIn = Software
[4] CertPathValidator : PKIX
                           ImplementedIn = Software
                           ValidationAlgorithm = draft-ietf-pkix-
new-part1-08.txt
[5] KeyPairGenerator : DSA
                           ImplementedIn = Software
                           KeySize = 1024
[6] CertificateFactory : X509|X.509
[7] AlgorithmParameterGenerator : DSA
                           ImplementedIn = Software
                           KeySize = 1024
[8] CertStore        : LDAP
                           ImplementedIn = Software
                           LDAPSchema = RFC2587
                       Collection
                           ImplementedIn = Software
[9] AlgorithmParameters : DSA
                           ImplementedIn = Software
[10] KeyFactory      : DSA
                           ImplementedIn = Software
[11] CertPathBuilder : PKIX
                           ImplementedIn = Software
                           ValidationAlgorithm = draft-ietf-pkix-
new-part1-08.txt
---------------------------------------------------------------
Provider[1]:: SunJSSE 1.41
Cryptographic Services::
[0] KeyStore         : PKCS12
[1] Signature        : MD5withRSA
                       SHA1withRSA
```

```
                        MD2withRSA
[2] TrustManagerFactory : SunX509
[3] KeyPairGenerator : RSA
[4] SSLContext       : SSL
                        SSLv3
                        TLS
                        TLSv1
[5] KeyManagerFactory : SunX509
[6] KeyFactory       : RSA
---------------------------------------------------------------
Provider[2]:: SunRsaSign 1.0
Cryptographic Services::
[0] Signature        : MD5withRSA
                        SHA1withRSA
                        MD2withRSA
[1] KeyPairGenerator : RSA
[2] KeyFactory       : RSA
---------------------------------------------------------------
Provider[3]:: SunJCE 1.4
Cryptographic Services::
[0] Cipher           : DES
                        Blowfish
                        TripleDES|DESede
                        PBEWithMD5AndTripleDES
                        PBEWithMD5AndDES
[1] KeyStore         : JCEKS
[2] KeyPairGenerator : DiffieHellman|DH
[3] AlgorithmParameterGenerator : DiffieHellman|DH
[4] AlgorithmParameters : TripleDES|DESede
                        PBEWithMD5AndDES|PBE
                        DES
                        Blowfish
                        DiffieHellman|DH
[5] KeyAgreement     : DiffieHellman|DH
[6] KeyGenerator     : HmacSHA1
                        TripleDES|DESede
                        HmacMD5
                        DES
                        Blowfish
[7] SecretKeyFactory : TripleDES|DESede
                        DES
                        PBEWithMD5AndDES
[8] KeyFactory       : DiffieHellman|DH
[9] Mac              : HmacMD5
                        HmacSHA1
---------------------------------------------------------------
Provider[4]:: SunJGSS 1.0
Cryptographic Services::
---------------------------------------------------------------
```

The output contains a wealth of information about various services supported by bundled providers. To interpret the results, follow the following simple rules:

- The left side of "`:`" has the service name and the right side has the algorithms or types supported.
- More than one algorithm or type name in the same line, separated by "`|`" imply aliases for the same name.
- Some of the entries have additional information in the form of name-value pairs. An example of such a name-value pair is "`ImplementedIn = Software`" for a number of entries.

Once you get comfortable with the output, you have figured out a lot about various cryptographic services available with J2SE SDK, v1.4. Regarding the supported services, the following observations are worth noting:

- Cipher service in provider "`SunJCE`" supports only symmetric algorithms. You cannot use this provider or any other bundled provider for public-key encryption.
- "`TripleDES`" and "`DESede`" are aliases for the same algorithm.
- Provider "`SUN`" has implementation for not only JCA services but also for a number of certificate validation services. We cover certificates and other related operations in Chapter 4, *PKI with Java*.
- Three different types of `KeyStore` are supported, each one in a different provider: "`JKS`" in "`SUN`", "`JCEKS`" in "`SunJCE`" and "`PKCS12`" and "`SunJSSE`".

If you are working with a J2SE v1.4 compliant environment from a vendor other than Sun or have installed third-party providers, the output may be different. In either case, **`crypttool`** is a good tool to explore your environment.

Installing and Configuring a Provider

Installing a provider means placing the jar file(s) having the provider classes at appropriate locations and modifying the security configuration files so that the application program is able to load and execute the provider class files. This can be done by installing the provider as a standard Java extension by placing the jar file in the *`jre-home`*`\lib\ext` directory where *`jre-home`* is the Java runtime software installation directory. If you have J2SE SDK, v1.4 installed in `c:\j2sdk1.4` then the *`jre-home`* will be `c:\j2sdk1.4\jre`. If you have only the JRE (**Java Runtime Environment**), then *`jre-home`* will be the root directory of the JRE installation, such as `c:\Program Files\Java\jre1.4.0`.

It is also possible to install a provider by just making the jar file available as a component in the `bootclasspath` of the program. This would require launching the client program with

the command **"java -Xbootclasspath/a:*provider-jar-file client-program-class*"**. Just setting the CLASSPATH to include the provider jar file doesn't work.

If the provider is not installed as an extension and it is to be accessed by a program where a Security Manager is installed, then it must be granted appropriate permissions in the global or user-specific policy file java.policy. Recall that JVM running an applet will most likely have a Security Manager installed. The syntax of policy files and other details on granting specific permissions are covered in Chapter 5, *Access Control*. However, the brief description given below would suffice for installing a provider.

The global policy file resides in the directory *jre-home*\lib\security. The default location of the user specific policy file is in the user home directory. A sample policy statement granting such permission to a provider with the name "MyJCE" and class files in myjce_provider.jar kept in directory c:\myjce appears below:

```
grant codeBase "file:/c:/myjce/myjce_provider.jar" {
  permission java.lang.RuntimePermission "getProtectionDomain";
  permission java.security.SecurityPermission
      "putProviderProperty.MyJCE";
};
```

After installation, a provider must be *configured* before it can be accessed by the client programs. This configuration is done either *statically* for the whole J2SE environment by modifying security properties file or *dynamically* for a given run of a program by invoking appropriate API calls from within the program.

Static configuration requires modification of the security properties file *jre-home*\lib\security\java.security. This file contains an entry for each provider, either bundled or installed, of the form

```
security.provider.n=master-class-name
```

Here *n* is a number specifying the priority, 1 being the highest, and *master-class-name* is the fully qualified name of the class in the provider jar file that extends the class java.security.Provider. To add a provider, simply insert an entry corresponding to the provider's master class with the appropriate priority number. Note that this may require some adjustment in the priority of existing providers.

For example, after installing Cryptix JCE provider (Cryptix JCE provider is an open source implementation available from http://www.cryptix.org) with lowest priority, a portion of the java.security file would look like:

```
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
security.provider.5=sun.security.jgss.SunProvider

#Added by Pankaj on July 22, 2002 for testing.
security.provider.6=cryptix.jce.provider.CryptixCrypto
```

If you have more than one provider with the same priority, then the registration by the last provider overrides the previous registrations. Also, you must not have gaps within the priority number sequence, otherwise only the providers with consecutive priorities starting at 1 are registered. If there is a typo in the fully qualified name of the master class then the corresponding provider is not registered. All this happens silently without any warning, so you must be careful while modifying the security properties file.

You can check for successful configuration by running the command **"crypttool listp"**. An invocation of this program should list all the providers, including the new ones, in the same order as the specified priority numbers. A frequent mistake, especially on development machines with multiple Java runtime software installed, is not realizing that the runtime environment of the **java** command may not be the same as the one you just configured. This is easily rectified by executing the command with **%JAVA_HOME%\bin\java** in place of **java** to launch the right JVM. Utility **crypttool** picks up the java executable from %JAVA_HOME%\bin directory, so make sure that the value of environment variable JAVA_HOME matches the Java installation you just configured.

A provider is dynamically configured within the client program code by calling addProvider or insertProviderAt method of Security class. For this to work, appropriate permission must be granted to the client code. For example, the following statement in the policy file java.policy provides the adequate permission to all code from directory c:\myclient.

```
grant codeBase "file:/c:/myclient/" {
  permission java.security.SecurityPermission
"insertProviderAt.*";
  permission java.security.SecurityPermission "addProvider.*";
};
```

Another thing to keep in mind is that the JCE engine authenticates the provider by verifying the signature on the code. The verification step looks for a signature by *JCE Code Signing CA* or a CA whose certificate has been signed by it. This is not a problem for bundled or commercial providers, as they are signed with appropriate private keys, but it becomes an issue with your own implementation of a JCE provider and most of the open source providers. For example, when I installed the "CryptixCrypto" provider and launched a program accessing one of its services, the exception java.security.NoSuchProviderException was thrown with a message saying: JCE cannot authenticate the provider CryptixCrypto.

If you do want to play with an unsigned provider during development, you can bypass the JCE engine by specifying an alternate JCE implementation. In the case of Cryptix provider, one way to do this is simply by removing the JDK's jce.jar file from *jre-home*\lib as the Cryptix provider comes with its own JCE classes.

Another option is to use the open source JCE provider from *Legion of the Bouncy Castle*, available from http://www.bouncycastle.org. This provider comes with an appropriate signed jar file and supports a wide variety of services and algorithms. For release 1.18 (the current one in

March 2003), you should download a file named `bcprov-jdk14-118.jar` and place it in *`jre-home`*`\lib\ext` directory and add the following line in your `java.security` file:

```
security.provider.6=org.bouncycastle.jce.provider.BouncyCastleProvider
```

If you do install this provider, run command **"`bin\crypttool listp`"** to get a list of active providers; and if this succeeds and shows `BC` as a provider, then command **"`bin\crypttool listp -provider BC -csinfo`"** to get a listing of available services and algorithms supported with this provider.

To recap, you must pay attention to the following while installing a security provider:

- The provider jar file has been placed in the standard extension directory or its path is specified through `–Xbootclasspath` argument to JVM.
- The provider jar has been granted appropriate permissions. This is required only if the program is running under a Security Manager. This is likely to be the case if your program is running within a container.
- An appropriate CA has signed the provider jar.

Why is installing a security provider so complicated? Compromise of a security provider can easily compromise all the security provided by cryptography. Hence, it is imperative that proper safeguards are in place. A number of the above mentioned steps are about ensuring that only trusted code is used as a security provider.

Why would someone want to use a third-party provider? Here are some good reasons:

- You need your Java application to be integrated into an existing environment that uses algorithms and/or types not supported by bundled providers.
- You bought special hardware to speed up your application but use of this hardware requires using the vendor's provider.
- The algorithms supported by the bundled providers are not strong enough for your requirements.
- You live in a country where you can only download the J2SE SDK with "limited" cryptography but want to use "unlimited" cryptography. We will talk more about this later in the section *Limited versus Unlimited Cryptography.*
- You invented a new algorithm or better implementation of an existing algorithm and want to use it.

Whether you use the provider supplied with J2SE v1.4.x SDK or install your own, the programs using the cryptographic services remain the same.

Cryptographic Keys

Secret keys, a stream of randomly generated bits appropriate for the chosen algorithm and purpose, are central to a number of cryptographic operations. In fact, much of the security offered

by cryptography depends on appropriate handling of keys, for the algorithms themselves are publicly published. What it means is that a key that can be easily compromised, computed, guessed, or found by trial and error with reasonable effort offers little or no security, no matter how secure the algorithm. Strength of security, or the degree of difficulty in determining the right key by a brute force exhaustive search, depends on the size and randomness of the key. For all these reasons, it is imperative that due diligence is exercised in selecting the right keys, using them properly and protecting them adequately.

However, not all cryptographic operations require secret keys. Certain operations work with a pair of keys—a private key that must be kept secret and a corresponding public key that can be shared freely.

The Java platform offers a rich set of abstractions, services and tools for generation, storage, exchange and use of cryptographic keys, simplifying the problem to careful use of these APIs and tools.

Java Representation of Keys

Java interface `java.security.Key` provides an *opaque*, algorithm and type independent representation of keys with the following methods:

`public String getAlgorithm()`

Returns the standard name of the algorithm associated with the key. Examples include `"DES"`, `"DSA"` and `"RSA"`, among many others.

`public byte[] getEncoded()`

Returns the encoded value of the key as a byte array or `null` if encoding is not supported. The type of encoding is obtained by method `getFormat()`. For `"RAW"` encoding format, the exact bytes comprising the key are returned. For `"X.509"` and `"PKCS#8"` format, the bytes representing the encoded key are returned.

`public String getFormat()`

Returns the encoding format for this key or null if encoding is not supported. Examples: `"RAW"`, `"X.509"` and `"PKCS#8"`.

As we know, there are two kinds of encryption algorithms: symmetric or secret key algorithms and asymmetric or public key algorithms. Symmetric algorithms use the same key for both encryption and decryption and it must be kept secret, whereas asymmetric algorithms use a pair of keys, one for encryption and another for decryption. These keys are represented by various subinterfaces of `Key` with self-explanatory names—`SecretKey`, `PrivateKey` and `PublicKey`. These are *marker* interfaces, meaning they do not have any methods and are used only for indicating the purpose and type-safety of the specific `Key` objects. Java Security API has many more `Key` subinterfaces that allow access of algorithm specific parameters, but they are rarely used directly in application programs and hence are not covered.

Generating Keys

A `Key` object is instantiated by either internal generation within the program or getting the underlying bit stream in some way from an external source such as secondary storage or another program. Let us look at how keys are generated programmatically.

A `SecretKey` for a specific algorithm is generated by invoking method `generateKey()` on `javax.crypto.KeyGenerator` object. `KeyGenerator` is an engine class implying that a concrete object is created by invoking the static factory method `getInstance()`, passing the algorithm name and optionally, the provider name as arguments. After creation, the `KeyGenerator` object must be initialized in one of two ways—algorithm independent or algorithm specific. Algorithm independent initialization requires only the key size in number of bits and an optional source of randomness. Here is example program `GenerateSecretKey.java` that generates a secret key for `DES` algorithm.

Listing 3-3 Generating a secret key

```
// File: src\jsbook\ch3\GenerateSecretKey.java
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.security.Key;

public class GenerateSecretKey {
  private static String formatKey(Key key){
    StringBuffer sb = new StringBuffer();
    String algo = key.getAlgorithm();
    String fmt = key.getFormat();
    byte[] encoded = key.getEncoded();
    sb.append("Key[algorithm=" + algo + ", format=" + fmt +
        ", bytes=" + encoded.length + "]\n");
    if (fmt.equalsIgnoreCase("RAW")){
      sb.append("Key Material (in hex):: ");
      sb.append(Util.byteArray2Hex(key.getEncoded()));
    }
    return sb.toString();
  }
  public static void main(String[] unused) throws Exception {
    KeyGenerator kg = KeyGenerator.getInstance("DES");
    kg.init(56); // 56 is the keysize. Fixed for DES
    SecretKey key = kg.generateKey();
    System.out.println("Generated Key:: " + formatKey(key));
  }
}
```

Running this program produces the following output:

```
C:\ch3\ex1>java GenerateSecretKey
Generated Key:: Key[algorithm=DES, format=RAW, bytes=8]
Key Material (in hex):: 10 46 8f 83 4c 8a 58 57
```

Run the same program again. Do you get the same key material? No, you get a different value. How is this explained? The `KeyGenerator` uses the default implementation of `SecureRandom` as a source of randomness and this generates a different number for every execution.

Generation of public and private key pair follows a similar pattern with class `KeyGenerator` replaced by `java.security.KeyPairGenerator` and method `SecretKey generateKey()` replaced by `KeyPair generateKeyPair()`. Example program `GenerateKeyPair.java` illustrates this.

Listing 3-4 Generating a public-private key pair

```
import java.security.KeyPairGenerator;
import java.security.KeyPair;
import java.security.PublicKey;
import java.security.PrivateKey;
import java.security.Key;

public class GenerateKeyPair {
  private static String formatKey(Key key){
    // Same as in GenerateSecretKey.java. hence omitted.
  }
  public static void main(String[] unused) throws Exception {
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
    kpg.initialize(512); // 512 is the keysize.
    KeyPair kp = kpg.generateKeyPair();
    PublicKey pubk = kp.getPublic();
    PrivateKey prvk = kp.getPrivate();
    System.out.println("Generated Public Key:: " +
formatKey(pubk));
    System.out.println("Generated Private Key:: " +
formatKey(prvk));
  }
}
```

Running this program produces:

```
C:\ch3\ex1>java GenerateKeyPair
Generated Public Key:: Key[algorithm=DSA, format=X.509, bytes=244]
Generated Private Key:: Key[algorithm=DSA, format=PKCS#8,
bytes=201]
```

Note that the format of public and private keys is not `RAW`. The public key is in `X.509` format and the private key is in `PKCS#8` format.

Utility **`crypttool`** has commands **`genk`** and **`genkp`** to generate secret keys and pairs of public-private keys, allowing the user to specify the algorithm, keysize and a way to save the generated keys. Refer to the section *Cryptography with `crypttool`* for more details.

Storing Keys

Keys need to be stored on secondary storage so that programs can access them conveniently and securely for subsequent use. This is accomplished through the engine class `java.security.KeyStore`. A `KeyStore` object maintains an in-memory table of key and certificate entries, indexed by *alias* strings, allowing retrieval, insertion and deletion of entries. This object can be initialized from a file and saved to a file. Such files are known as keystore files. For security reasons, keystore files and, optionally, individual entries, are password protected.

The following code fragment illustrates initializing a `KeyStore` object from a JCEKS keystore file `test.ks` protected with password `"changeit"`.

```
FileInputStream fis = new FileInputStream("test.ks");
KeyStore ks = KeyStore.getInstance("JCEKS");
ks.load(fis, "changeit".toCharArray());
```

Different providers or even the same provider supporting different keystore types can store keys in different types of persistent store: a flat file, a relational database, an LDAP (**Lightweight Data Access Protocol**) server or even MS-Windows Registry.

J2SE v1.4 bundled providers support flat file formats `JKS` and `JCEKS`. `JKS` keystore can hold only private key and certificate entries whereas `JCEKS` keystore can also hold secret key entries. There is also read-only support for keystore type `PKCS12`, allowing import of Netscape and MSIE browser certificates into a Java keystore

Java keystore types `JKS` and `JCEK` work okay for development and simple applications with small number of entries, but may not be suitable in the production environment that is required to support a large number of entries. Consider investing in a commercial provider for such uses.

Java platform includes a simple command line utility **`keytool`** to manage keystores. The primary purpose of this tool is to generate public and private key pairs and manage certificates for PKI based applications. We talk more about this tool in Chapter 4, *PKI with Java*.

Encryption and Decryption

Encryption is the process of converting normal data or plaintext to something incomprehensible or cipher-text by applying mathematical transformations. These transformations are known as encryption algorithms and require an encryption key. Decryption is the reverse process of getting back the original data from the cipher-text using a decryption key. The encryption key and the decryption key could be the same as in symmetric or secret key cryptography, or different as in asymmetric or public key cryptography.

Algorithms

A number of encryption algorithms have been developed over time for both symmetric and asymmetric cryptography. The ones supported by the default providers in J2SE v1.4 are: DES,

TripleDES, Blowfish, PBEWithMD5AndDES, and PBEWithMD5AndTripleDES. Note that these are all symmetric algorithms.

DES keys are 64 bits in length, of which only 56 are effectively available as one bit per byte is used for parity. This makes DES encryption quite vulnerable to brute force attack. TripleDES, an algorithm derived from DES, uses 128-bit keys (112 effective bits) and is considered much more secure. Blowfish, another symmetric key encryption algorithm, could use any key with size up to 448 bits, although 128-bit keys are used most often. Blowfish is faster than TripleDES but has a slow key setup time, meaning the overall speed may be less if many different keys are used for small segments of data. Algorithms PBEWithMD5AndDES and PBEWithMD5AndTripleDES take a password string as the key and use the algorithm specified in PKCS#5 standard.

There are currently four FIPS approved symmetric encryption algorithms: DES, TripleDES, AES (**Advanced Encryption Standard**) and Skipjack. You can find more information about these at http://csrc.nist.gov/CryptoToolkit/tkencryption.html. Among these, AES is a new standard and was approved only in 2001. Note that both AES and Skipjack are not supported in J2SE v1.4.[1]

All these algorithms operate on a block of data, typically consisting of 64 bits or 8 bytes, although smaller blocks are also possible. Each block can be processed independently or tied to the result of processing on the earlier block, giving rise to different *encryption modes*. Commonly used and supported modes include ECB (**Electronic CookBook)** mode, whereby each block is processed independently, CBC (**Cipher Block Chaining**) mode, whereby the result of processing the current block is used in processing the next block), CFB (**Cipher Feed Back**) and OFB (**Output Feed Back**). Detailed information on these modes and their performance, security and other characteristics can be found in the book *Applied Cryptography* by noted cryptographer Bruce Schneier.

CFB and OFB modes allow processing with less than 64 bits, with the actual number of bits, usually a multiple of 8, specified after the mode such as CFB8, OFB8, CFB16, OFB16 and so on. When a mode requires more than 1 byte to do the processing, such as ECB, CBC, CFB16, OFB16 and so on, the data may need to be padded to become a multiple of the block size. Bundled providers support PKCS5Padding, a padding scheme specified in PKCS#5. Also, modes CBC, CFB and OFB need an 8-byte *Initialization Vector*, so that even the first block has an input to start with. This must be same for both encryption and decryption.

Java API

Java class `javax.crypto.Cipher` is the engine class for encryption and decryption services. A concrete `Cipher` object is created by invoking the static method `getInstance()` and requires a transform string of the format *`algorithm/mode/padding`* (an example string would be `"DES/ECB/PKCS5Padding"`) as an argument. After creation, it must be initialized with the key and, optionally, an initialization vector. After initialization, method `update()` can be called any number of times to pass byte arrays for encryption or decryption, terminated by a `doFinal()` invocation.

1. Support for AES has been added to J2SE v1.4.2.

The example program `SymmetricCipherTest.java` illustrates symmetric encryption and decryption. This program generates a secret key for DES algorithm, encrypts the bytes corresponding to a string value using the generated key and finally decrypts the encrypted bytes to obtain the original bytes. Note the use of an initialization vector for both encryption and decryption. Although the code in this program works on a byte array, it is possible to pass multiple smaller chunks of byte sequences to the `Cipher` instance before initiating the encryption or decryption.

The code presented here doesn't list individual exceptions thrown in method `encrypt()` and `decrypt()`, but you can find them in the electronic version of the source file.

Listing 3-5 Encryption and Decryption with a symmetric Cipher

```
// File: src\jsbook\ch3\SymmetricCipherTest.java
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.Cipher;

public class SymmetricCipherTest {
  private static byte[] iv =
      { 0x0a, 0x01, 0x02, 0x03, 0x04, 0x0b, 0x0c, 0x0d };

  private static byte[] encrypt(byte[] inpBytes,
      SecretKey key, String xform) throws Exception {
    Cipher cipher = Cipher.getInstance(xform);
    IvParameterSpec ips = new IvParameterSpec(iv);
    cipher.init(Cipher.ENCRYPT_MODE, key, ips);
    return cipher.doFinal(inpBytes);
  }

  private static byte[] decrypt(byte[] inpBytes,
      SecretKey key, String xform) throws Exception {
    Cipher cipher = Cipher.getInstance(xform);
    IvParameterSpec ips = new IvParameterSpec(iv);
    cipher.init(Cipher.DECRYPT_MODE, key, ips);
    return cipher.doFinal(inpBytes);
  }

  public static void main(String[] unused) throws Exception {
    String xform = "DES/ECB/PKCS5Padding";
    // Generate a secret key
    KeyGenerator kg = KeyGenerator.getInstance("DES");
    kg.init(56); // 56 is the keysize. Fixed for DES
    SecretKey key = kg.generateKey();

    byte[] dataBytes =
```

```
        "J2EE Security for Servlets, EJBs and Web
Services".getBytes();

    byte[] encBytes = encrypt(dataBytes, key, xform);
    byte[] decBytes = decrypt(encBytes, key, xform);

    boolean expected = java.util.Arrays.equals(dataBytes,
decBytes);
    System.out.println("Test " + (expected ? "SUCCEEDED!" :
"FAILED!"));
  }
}
```

Compiling and running this program is similar to other programs in this chapter.

Encryption algorithm PBEWithMD5AndDES requires a slightly different initialization sequence of the Cipher object. Also, there is an alternate mechanism to do encryption decryption involving classes `CipherInputStream` and `CipherOutputStream`. We do not cover these methods here. If you are interested in their use, look at the source code of utility **`crypttool`**.

The same sequence of calls, with appropriate modifications, would be valid for asymmetric cryptography as well. The example program `AsymmetricCipherTest.java` illustrates this.

Listing 3-6 Encryption and Decryption with a asymmetric Cipher

```
// File: src\jsbook\ch3\AsymmetricCipherTest.java
import java.security.KeyPairGenerator;
import java.security.KeyPair;
import java.security.PublicKey;
import java.security.PrivateKey;
import javax.crypto.Cipher;

public class AsymmetricCipherTest {
  private static byte[] encrypt(byte[] inpBytes, PublicKey key,
      String xform) throws Exception {
    Cipher cipher = Cipher.getInstance(xform);
    cipher.init(Cipher.ENCRYPT_MODE, key);
    return cipher.doFinal(inpBytes);
  }
  private static byte[] decrypt(byte[] inpBytes, PrivateKey key,
      String xform) throws Exception{
    Cipher cipher = Cipher.getInstance(xform);
    cipher.init(Cipher.DECRYPT_MODE, key);
    return cipher.doFinal(inpBytes);
  }

  public static void main(String[] unused) throws Exception {
```

```
        String xform = "RSA/NONE/PKCS1PADDING";
        // Generate a key-pair
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(512); // 512 is the keysize.
        KeyPair kp = kpg.generateKeyPair();
        PublicKey pubk = kp.getPublic();
        PrivateKey prvk = kp.getPrivate();

        byte[] dataBytes =
            "J2EE Security for Servlets, EJBs and Web
    Services".getBytes();

        byte[] encBytes = encrypt(dataBytes, pubk, xform);
        byte[] decBytes = decrypt(encBytes, prvk, xform);

        boolean expected = java.util.Arrays.equals(dataBytes,
    decBytes);
        System.out.println("Test " + (expected ? "SUCCEEDED!" :
    "FAILED!"));
      }
    }
```

Note that this program uses a `KeyPairGenerator` to generate a public key and a private key. The public key is used for encryption and the private key is used for decryption. As there is no padding, there was no need to have an initialization vector.

The only caveat is that J2SE v1.4 bundled providers do not support asymmetric encryption algorithms. You would need to install a third-party JCE provider for this. You could use the Bouncy Castle provider. In fact, the above program is tested against this provider.

Message Digest

Message digests, also known as *message fingerprints* or *secure hash*, are computed by applying a one-way hash function over the data bits comprising the message. Any modification in the original message, either intentional or unintentional, will *most certainly* result in a change of the digest value. Also, it is computationally impossible to derive the original message from the digest value. These properties make digests ideal for detecting changes in a given message. Compute the digest before storing or transmitting the message and then compute the digest after loading or receiving the message. If the digest values match then one can be sure with good confidence that the message has not changed. However, this scheme fails if a malicious interceptor has access to both the original message and its digest. In this case the interceptor could easily alter the message, compute the digest of the modified message and replace the original digest with the new one. The solution, as we see in the next section, is to secure the message digest by encrypting it with a secret key.

A common use of message digests is to securely store and validate passwords. The basic idea is that you never store the password in clear-text. Compute the message digest of the password and store the digest value. To verify the password, compute its digest and match it with the stored value. If both values are equal, the verification succeeds. This way no one, not even the administrator, gets to know your password. A side effect of this mechanism is that you cannot get back a forgotten password. This is not really as bad as it sounds, for you can always get it changed to a temporary password by an administrator, and then change it to something that only you know.

Message digests of messages stored in byte arrays are computed using engine class `java.security.MessageDigest`. The following program illustrates this.

Listing 3-7 Computing message digest

```
// File: src\jsbook\ch3\ComputeDigest.java
import java.security.MessageDigest;
import java.io.FileInputStream;

public class ComputeDigest {
  public static void main(String[] unused) throws Exception{
  String datafile = "ComputeDigest.java";

  MessageDigest md = MessageDigest.getInstance("SHA1");
  FileInputStream fis = new FileInputStream(datafile);
  byte[] dataBytes = new byte[1024];
  int nread = fis.read(dataBytes);
  while (nread > 0) {
    md.update(dataBytes, 0, nread);
    nread = fis.read(dataBytes);
  };
  byte[] mdbytes = md.digest();
  System.out.println("Digest(in hex):: " +
Util.byteArray2Hex(mdbytes));
  }
}
```

A concrete, algorithm-specific `MessageDigest` object is created following the general pattern of all engine classes. The invocation of `update()` method computes the digest value and the `digest()` call completes the computation. It is possible to make multiple invocations of `update(byte[] bytes)` before calling the `digest()` method, thus avoiding the need to accumulate the complete message in a single buffer, if the original message happens to be fragmented over more than one buffer or cannot be kept completely in main memory. This is likely to be the case if the data bytes are being read from a huge file in fixed size buffers. In fact, convenience classes `DigestInputStream` and `DigestOutputStream`, both in the package `java.security`, exist to compute the digest as the bytes flow through the associated streams.

The verification or check for integrity of the message is done by computing the digest value and comparing this with the original digest for size and content equality. Class `MessageDigest` even includes static method `isEqual(byte[] digestA, byte[] digestB)` to perform this task.

Theoretically, because a much larger set of messages get mapped to a much smaller set of digest values, it is possible that two or more messages will have the same digest value. For example, the set of 1 KB messages has a total of $2^{(8*1024)}$ distinct messages. If the size of the digest value is 128 then there are only 2^{128} different digest values possible. What it means is that there are, on the average, $2^{(8*1024—128)}$ different 1KB messages with the same digest value. However, a brute-force search for a message that results in a given digest value would still require examining, on the average, 2^{127} messages. The problem becomes a bit simpler if one were to look for *any* pair of messages that give rise to the same digest value, requiring, on the average, only 2^{64} attempts. This is known as the *birthday attack*, deriving its name from a famous mathematics puzzle, whose result can be stated as: there is more than a 50 percent chance that you will find someone with the same birthday as yours in a party of 183 persons. However, this number drops to 23 for *any* pair to have the same day as their birthday.

The providers bundled with J2SE v1.4 support two message digest algorithms: SHA (**Secure Hash Algorithm**) and MD5. SHA, also known as SHA-1, produces a message digest of 160 bits. It is a FIPS (**Federal Information Processing Standard**) approved standard. In August 2002, NIST announced three more FIPS approved standards for computing message digest: SHA-256, SHA-384 and SHA-512. These algorithms use a digest value of 256, 384 and 512 bits respectively, and hence provide much better protection against brute-force attacks. MD5 produces only 128 bits as message digest, and is considerably weaker.

Message Authentication Code

Message Authentication Code or MAC is obtained by applying a secret key to the message digest so that only the holder of the secret key can compute the MAC from the digest and hence, the message. This method thwarts the threat posed by a malicious interceptor who could modify the message and replace the digest with the digest of the modified message, for the interceptor won't have access to the secret key. Of course, there has to be a secure way to share the secret key between the sender and the recipient for this to work.

J2SE includes class `javax.crypto.Mac` to compute MAC. This class is somewhat similar to the `MessagDigest` class, except for the following:

- A `Mac` object must be initialized with a secret key.
- There is method `doFinal()` in place of `digest()`.

Another difference between classes for MAC and message digest is that there are no `MacInputStream` and `MacOutputStream` classes.

The example program to illustrate MAC computation is similar to the one for Message Digest.

Listing 3-8 Computing Message Authentication Code (MAC)

```
// File: src\jsbook\ch3\ComputeMAC.java
import javax.crypto.Mac;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.io.FileInputStream;

public class ComputeMAC {
  public static void main(String[] unused) throws Exception{
    String datafile = "ComputeDigest.java";

    KeyGenerator kg = KeyGenerator.getInstance("DES");
    kg.init(56); // 56 is the keysize. Fixed for DES
    SecretKey key = kg.generateKey();

    Mac mac = Mac.getInstance("HmacSHA1");
    mac.init(key);

    FileInputStream fis = new FileInputStream(datafile);
    byte[] dataBytes = new byte[1024];
    int nread = fis.read(dataBytes);
    while (nread > 0) {
      mac.update(dataBytes, 0, nread);
      nread = fis.read(dataBytes);
    };
  byte[] macbytes = mac.doFinal();
  System.out.println("MAC(in hex):: " +
Util.byteArray2Hex(macbytes));
  }
}
```

J2SE bundled providers support MAC algorithms HmacSHA1 and HmacMD5, corresponding to message digest algorithms SHA1 and MD5.

Digital Signature

Encrypting the digest of a message with the private key using asymmetric cryptography creates the digital signature of the person or entity known to own the private key. Anyone with the corresponding public key can decrypt the signature to get the message digest and verify that the message digest indeed corresponds to the original message and be confident that it must have been encrypted with the private key corresponding to the public key. As the private key is not made

public, it can be deduced that the message was signed by the owner of the private key. Generally, these are the same properties as the ones associated with a signature on paper.

Note that use of a digital signature requires a digest algorithm and an asymmetric encryption algorithm.

Algorithms

Currently, there are three FIPS-approved digital signature algorithms: DSA, RSA and ECDSA (**Elliptic Curve Digital Signature Algorithm**). More information on these algorithms can be found at http://csrc.nist.gov/CryptoToolkit/tkhash.html.

Java API

Java class `java.security.Signature` represents the signature service and has methods to create and verify a signature. Like any engine class, a concrete Signature object is created by invoking the static method `getInstance()`. For signing data bytes, it must be initialized using `initSign()` with the private key as an argument. A subsequent signature creation operated, through the method `sign()`, produces the signature bytes. Similarly, the verification operation, through the method `verify()`, after initialization using `initVerify()` with the public key as the argument, verifies whether a particular signature has been created using the corresponding private key or not.

The example program `SignatureTest.java` illustrates signing and verification.

Listing 3-9 Signature creation and verification

```
// File: src\jsbook\ch3\ex1\SignatureTest.java
import java.security.KeyPairGenerator;
import java.security.KeyPair;
import java.security.PublicKey;
import java.security.PrivateKey;
import java.security.Signature;
import java.io.FileInputStream;

public class SignatureTest {
  private static byte[] sign(String datafile, PrivateKey prvKey,
      String sigAlg) throws Exception {
    Signature sig = Signature.getInstance(sigAlg);
    sig.initSign(prvKey);
    FileInputStream fis = new FileInputStream(datafile);
    byte[] dataBytes = new byte[1024];
    int nread = fis.read(dataBytes);
    while (nread > 0) {
      sig.update(dataBytes, 0, nread);
      nread = fis.read(dataBytes);
    };
```

```
    return sig.sign();
  }
  private static boolean verify(String datafile, PublicKey pubKey,
      String sigAlg, byte[] sigbytes) throws Exception {
    Signature sig = Signature.getInstance(sigAlg);
    sig.initVerify(pubKey);
    FileInputStream fis = new FileInputStream(datafile);
    byte[] dataBytes = new byte[1024];
    int nread = fis.read(dataBytes);
    while (nread > 0) {
      sig.update(dataBytes, 0, nread);
      nread = fis.read(dataBytes);
    };
    return sig.verify(sigbytes);
  }
  public static void main(String[] unused) throws Exception {
    // Generate a key-pair
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
    kpg.initialize(512); // 512 is the keysize.
    KeyPair kp = kpg.generateKeyPair();
    PublicKey pubk = kp.getPublic();
    PrivateKey prvk = kp.getPrivate();

    String datafile = "SignatureTest.java";
    byte[] sigbytes = sign(datafile, prvk, "SHAwithDSA");
    System.out.println("Signature(in hex):: " +
        Util.byteArray2Hex(sigbytes));

    boolean result = verify(datafile, pubk, "SHAwithDSA",
sigbytes);
    System.out.println("Signature Verification Result = " +
result);
  }
}
```

Besides SHAwithDSA, the J2SE bundled providers support SHA1withRSA, MD5withRSA and MD2with RSA signature algorithms.

Key Agreement

Secure exchange of data over an insecure channel requires the data packets to be encrypted by the sender and decrypted by the receiver. In such a scenario, one could use symmetric cryptography for encryption and decryption but that would require the communicating parties to use the same secret key. This is not viable for an open communication medium like the Internet that must allow secure exchange among unknown parties without prior agreement to share secret keys.

One might think that public key cryptography is ideally suited to solve this problem. The sender would do the encryption using the public key of the recipient and the recipient would decrypt the message using its own private key. The whole scheme would only require each party to have or generate its own key pair and share the public key with others.

In practice, this approach has a small problem. The performance overhead of public key encryption and decryption is unacceptably high. However, there is a solution to this problem, for the performance issue can be addressed by generating a secret key for encrypting the actual data and encrypting the secret key with the public key. The recipient could now use his or her private key to decrypt the secret key and then use this key to decrypt the data using much faster symmetric decryption.

Even this scheme requires that every entity must have the public key of all other entities with whom it wishes to communicate. This precondition will preclude secure communication between parties that do not know each other beforehand.

One solution is to use the public key cryptography and a key agreement mechanism to agree upon a secret key in such a way that the key itself is never transmitted and cannot be intercepted or deduced from the intercepted traffic. Once such a secret key is agreed upon, it can be used for data encryption and decryption.

J2SE v1.4 supports key agreement operations through the service class `javax.crypto.KeyAgreement`. We do not get into the programmatic usage details of this class but instead look at one of the key agreement algorithms supported by J2SE v1.4—Diffie-Hellman.

1. The initiator generates a public and private key pair and sends the public key, along with the algorithm specification, to the other party.
2. The other party generates its own public and private key pair using the algorithm specification and sends the public key to the initiator.
3. The initiator generates the secret key using its private key and the other party's public key.
4. The other party also generates the secret key using its private key and the initiator's public key. Diffie-Hellamn algorithm ensures that both parties generate the same secret key.

This sequence of steps is pictorially illustrated in *Figure 3-2*.

As we see in Chapter 6, *Securing the Wire*, this mechanism is used by SSL to agree upon a shared secret key and secure the exchange of data.

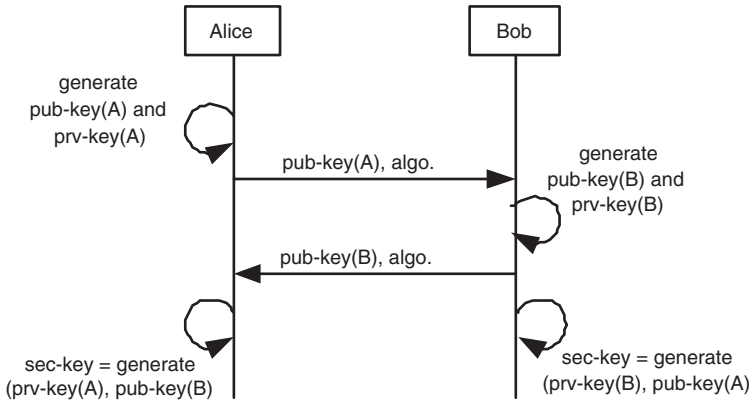

Figure 3-2 Diffie-Hellman Key Agreement.

Summary of Cryptographic Operations

We have covered a number of cryptographic operations, their characteristics and uses in the previous sections. Let us recap the main points with help of *Figure 3-3*.

Here is the basic scenario: Alice has some data she wants to share with Bob. Depending upon the situation, she could use one or more of the cryptographic operations discussed in this chapter, as explained below.

1. **Ensure that the data has not been accidentally corrupted.** Alice computes the digest of the data and sends the digest value along with the message. Bob, after receiving the data, computes the digest and matches it against the received value. A successful match implies that the data is not corrupted.
2. **Ensure that the data has not been maliciously modified.** In this case, Alice cannot rely on digest value, as the malicious middleman could simply replace the digest value after modifying the data. So, she arranges to share a secret key with Bob and uses this key to compute the MAC of the data. Not being in the possession of the secret key, the middleman now cannot replace the MAC.
3. **Ensure that the data remains confidential.** Alice shares a secret key with Bob and uses this key to encrypt the data with a symmetric encryption algorithm.
4. **Ensure that the data remains confidential but without a shared secret key.** Alice has Bob's public key. She uses this key to encrypt the data. Bob decrypts it using his private key.
5. **Prove to Bob that the data has come from Alice.** Alice uses her private key to sign the data. Bob can verify the signature using her public key and be sure that the data

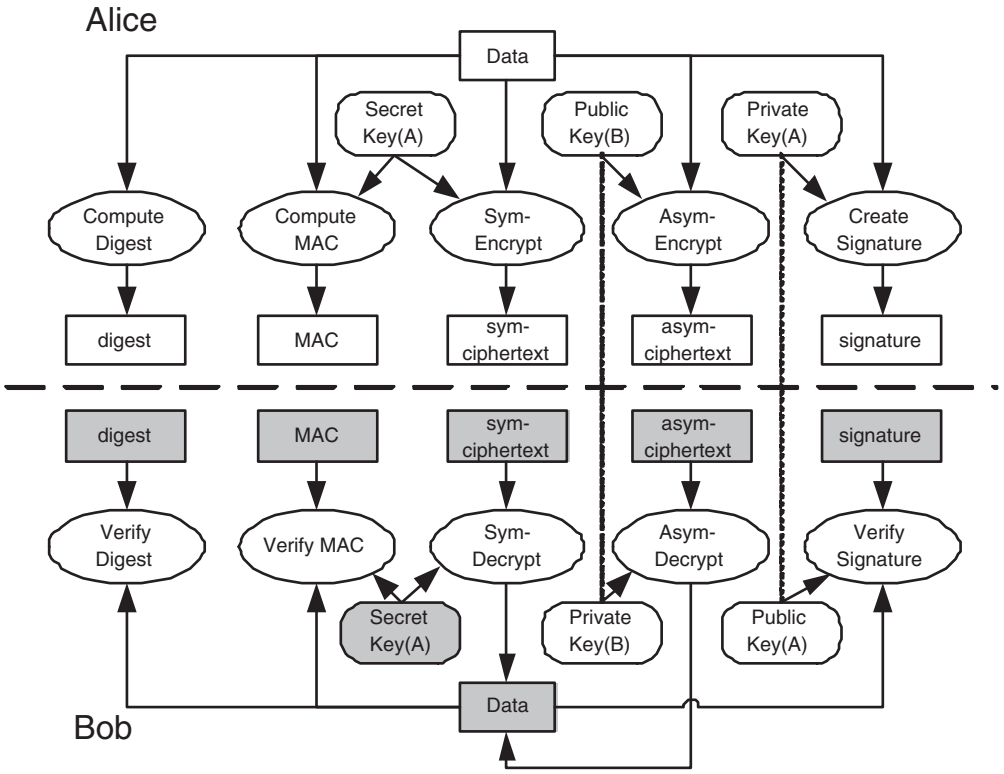

Figure 3-3 Cryptographic operations and their uses.

indeed originated from Alice. This also guarantees that the data has not been modified in transit.

6. **Prove to Bob that the data came from Alice and keep it confidential.** Alice signs the data with her private key and then encrypts it using Bob's public key. Bob decrypts it using his private key and verifies the signature using Alice's public key.

J2SE SDK includes classes to carry out these operations programmatically but includes no ready-made tool. As mentioned earlier, JSTK software includes a tool called **`cryptool`** written using these classes. The next section talks about this tool. You may find it useful to experiment with different input values for cryptographic operation, keys, algorithm and input data.

Cryptography with `crypttool`

While writing this book, I strongly felt the need for such a tool that could allow me to carry out the cryptographic operations without writing code and not finding one, wrote **`crypttool`**, a command line utility to carry out common cryptographic operations. This tool is patterned after **`keytool`**, a command line utility bundled with J2SE for generating public and private key pairs and managing certificates. In a number of ways, **`crypttool`** complements **`keytool`** by providing additional functionality to a Java developer.

Here is a brief description of how **`crypttool`** operates: it accepts a command name and other input as command line options, carries out the operation and writes the result to the standard output. You can get a listing of all the supported commands with brief descriptions by invoking **"`crypttool help`"**. Command specific options and other details can be displayed by executing **"`crypttool <cmd> help`"**. Refer to the appendix *JSTK Tools* for a complete listing of commands and their options.

Let us have a brief session with **`crypttool`**. This also gives us an opportunity to recapitulate all the operations that we have covered in this chapter.

The first step is to generate a secret key using `DESede` or `TripleDES` algorithm and save the generated key in a file. This key will later be used for encryption/decryption and computing the MAC of data in a file.

```
C:\jstk>bin\crypttool genk -action save -file test.sk \
-algorithm DESede -keysize 112
SecretKey written to file: test.sk
```

Execution of this command takes a while—around 7 seconds on a 900 MHz. AMD Athlon machine running Windows 2000. This is primarily due to initialization overhead for secure random number generator. J2SE v1.4.1 has optimized this step for the Windows platform and the corresponding time is less than 4 seconds, which is still noticeably slow. Get used to the slow start for standalone programs that use cryptography. Thankfully, the subsequent operations in the same JVM are much quicker.

What gets written to the file `test.sk` is not the raw bits of the generated key but the serialized Java object of type `SecretKey`. If you want to see the raw bits in hex format, use—**`action print`** instead of –**`action save`** in the **`crypttool`** invocation. You may also want to try other algorithms and keysizes supported in your environment.

Let us use the generated key to encrypt a file and then decrypt the encrypted file. We will use `DESede` algorithm in CFB8 mode, matching the algorithm of key generation.

```
C:\jstk>bin\crypttool crypt -op enc -infile build.xml –outfile \
test.enc -keyfile test.sk -iv 12345678 -transform \
DESede/CFB8/NoPadding
encrypted file "build.xml" to "test.enc"

C:\jstk>bin\crypttool crypt -op dec -infile test.enc \
–outfile test.dec \
```

```
-keyfile test.sk -iv 12345678 -transform \
DESede/CFB8/NoPadding
decrypted file "test.enc" to "test.dec"
```

Note that padding is not required as CFB8 mode operates on 8-bit blocks. Other modes such as CFB32 or CBC would require a padding scheme such as PKCS5Padding. Also note the use of initialization vector by specifying –**iv** option. The string specified as the initialization vector is converted into a byte array by **crypttool**. If you do not specify an initialization vector for encryption then the underlying implementation will supply one. You must specify the same value for decryption.

Le us compare the decrypted file with the original file.

```
C:\jstk>comp test.dec build.xml
Comparing test.dec and build.xml...
Files compare OK
```

As expected, the decryption retrieves the original content.

You could use this command to encrypt sensitive information on disk or attachments that must be sent over the Internet. In such situations, use of a secret key, an initialization vector and a specific transformation scheme could be cumbersome. A better method is to use a password based encryption as per PKCS#5 standard. This is supported by Java and also by **crypttool**. Just replace the –**keyfile**, **-iv** and –**transform** options with –**password** option followed by the password. By default, PBEWithMD5AndDES algorithm is used for encryption and decryption.

```
C:\jstk>bin\crypttool crypt -op enc -infile build.xml \
–outfile test.enc -password changeit
encrypted file "build.xml" to "test.enc"

C:\jstk>bin\crypttool crypt -op dec -infile test.enc \
–outfile test.dec -password changeit
decrypted file "test.enc" to "test.dec"
```

Our next task is to generate a key pair, use the private key to sign a document and the public key to verify the signature.

```
C:\jstk>bin\crypttool genkp -action save -file test.kp
KeyPair written to file: test.kp

C:\>crypttool sign -infile build.xml -keyfile test.kp \
-sigfile build.sig
signature written to file: build.sig

C:\>crypttool sign -verify -infile build.xml \
–keyfile test.kp -sigfile build.sig
verification succeeded
```

By default, **crypttool** uses the DSA algorithm with a key size of 512 bits to generate key pair and SHA1withDSA algorithm for signature. Also, it knows to pick the private key for signature creation and public key for verification from a file having a serialized KeyPair object. In a real application, though, one would keep them in separate files and in a format understood by widely used programs. The private and public keys are typically stored in a keystore with the private key protected by a password and the public key embedded in a certificate. **crypttool** can pick up private and public key from a keystore as well.

Use of **crypttool** to compute message digest and MAC is left as an exercise.

Limited versus Unlimited Cryptography

When you download and install J2SE v1.4, by default you get cryptographic capabilities that are termed as *strong* but *limited* by *Java Cryptographic Extension Reference Guide*. What does it mean? Here the term "strong" means that cryptographic algorithms that are considered cryptographically hard to break, such as TripleDES, RSA and so on. are supported. The term "limited" means that the keysize supported by these algorithms is limited to certain values.

A jurisdiction policy file controls the keysize supported. We learn more about policy files in Chapter 5, *Access Control*. The brief summary is that the code invoking the cryptographic algorithms checks the policy file before continuing the operation. This policy file itself resides in a signed jar file, and cannot be modified without detection.

These policy files reside in jar files local_policy.jar and US_export_policy.jar, both located in *jre-home*\lib\security directory. The policy file default_US_export.policy, archived within US_export_policy.jar, specifies the permissions allowed by US export laws. This includes all the cryptographic classes packaged within J2SE v1.4.x. Policy file default_local.policy, archived within local_policy.jar, specifies permissions that can be freely imported worldwide. Let us look at this file.

```
// File: default_local.policy
// Some countries have import limits on crypto strength.
// This policy file is worldwide importable.
grant {
    permission javax.crypto.CryptoPermission "DES", 64;
    permission javax.crypto.CryptoPermission "DESede", *;
    permission javax.crypto.CryptoPermission "RC2", 128,
                  "javax.crypto.spec.RC2ParameterSpec", 128;
    permission javax.crypto.CryptoPermission "RC4", 128;
    permission javax.crypto.CryptoPermission "RC5", 128,
          "javax.crypto.spec.RC5ParameterSpec", *, 12, *;
    permission javax.crypto.CryptoPermission "RSA", 2048;
    permission javax.crypto.CryptoPermission *, 128;
};
```

If you are within the U.S. and want to use a larger keysize, you can download JCE Unlimited Strength Jurisdiction Policy files from Sun's J2SE download page and replace the default policy jar files in *jre-home*\lib\security directory with downloaded jar files. The default_local.policy file for unlimited strength is shown below.

```
// File: default_local.policy
// Country-specific policy file for countries with no limits on
// crypto strength.
grant {
    // There is no restriction to any algorithms.
    permission javax.crypto.CryptoAllPermission;
};
```

The jar files having these policy files are signed and hence cannot be modified without detection.

A brief overview of legal issues associated with cryptography can be found in the section *Legal Issues with Cryptography*, on page 79.

Performance of Cryptographic Operations

Cryptographic operations are compute-intensive and do have an impact on overall application performance. However, not all operations and, for a given operation, all algorithms use the same number of CPU cycles for each unit of data processed. In fact, when selecting a particular algorithm for an application, speed of processing is an important criterion.

Table 3-2 lists the encryption and decryption rate (in Kbytes per second) for a number of algorithms. These measurements were taken on a 900MHz AMD Athlon machine running Windows 2000 and Sun's J2SE v1.4 JVM in server mode using repeated processing of a large (more than 1 MB) text file. The time spent in I/O and initialization and a few minutes of JVM warmup is not included in the reported figures.

Table 3-2 Encryption/Decryption Performance Measurements

Transformation, Keysize	Encryption Rate (KBytes/sec)	Decryption Rate (KBytes/sec)
DES/CBC/PKCS5Padding, 56 bits	2720	2302
TripleDES/ECB/PKCS5Padding, 112 bits	1080	1010
Blowfish, 128 bits	5090	3010
PBEWithMD5AndDES	2660	2270

These figures indicate that Blowfish is the fastest among all the reported algorithms. Interestingly, the decryption is significantly slower than encryption with Blowfish.

How about signature creation and verification performance? Table 3-3 has the measurement figures for signing and verifying the same document.

Table 3-3 Signature Creation/Verification Performance

Algorithm, Keysize	Signing Rate (KBytes/sec)	Verification Rate (KBytes/sec)
`SHA1WithDSA`, 512 bits	12080	11890
`SHA1WithDSA`, 1024 bits	11780	11580
`SHA1WithRSA`, 512 bits	16950	16910
`SHA1WithRSA`, 1024 bits	16070	16000

It is quite obvious that signing and verifying are significantly faster than encryption and decryption operations. Also, `SHA1WithRSA` is almost one and a half times faster than `SHA1WithDSA`.

These measurements are taken with the **`"crypttool bench"`** command. Use it within your environment to compare different algorithms and estimate crypto overhead for your application.

There are many ways to speed up the performance of these operations. A commonly used mechanism, especially for large volume applications, is to use special cryptographic accelerator cards. As most of the cryptographic algorithms can have extremely efficient hardware-based implementations, an order of magnitude improvement is not uncommon.

Practical Applications

Now that we have looked at most of the basic cryptographic services and have an idea of how they work, let us ask this question: What good are they? What can they do for us? As we have been saying all along, despite the abstract nature, cryptography is quite useful and can do pretty mighty things.

Confidentiality. Encrypted information is virtually hidden from everyone who doesn't know how to decrypt it or doesn't have the appropriate key. This makes it possible to share secret information over insecure communication channels, such as the Internet, thus providing confidentiality even though the network itself is quite open. The same applies to data stored on disk. Encryption ensures confidentiality of stored data even if the computer itself gets compromised or stolen.

Integrity. There are times when you want to detect intentional tampering or unintentional corruption of data. This goal can be achieved by computing the digest value of the original and the current data. A mismatch would indicate some sort of change in the data. If the threat of intentional tampering exists for both the data and the digest value then MAC can be used as the detection mechanism.

Non-repudiation. A physical signature on paper, along with the visually observable state of the paper, proves the authenticity of the document and is legally binding. Public key cryptography-based digital signature performs the same role for electronic documents.

Although these are quite powerful capabilities, in reality, things are more complex. Passwords are prone to be easily guessed or to be captured by tricks or "stolen" by social engineering. The use of a private key by a computer program is not always same as the use by the stated owner of the key. A compromised computer can trick a human user into doing things that the user may never have done knowingly. Finally, the cryptography itself is not fully resistant to attacks. Someone with good skill, sufficient determination and ample computing power can defeat most cryptographic protection.

But before we proceed to dismiss cryptography as useless junk, let us think about the physical world. Every now and then, the best-kept secrets become "public" due to carelessness or malicious intent of the parties in the know. Cases of forged documents or signatures are not unheard of. Even the most wary are not immune from being duped by con artists. All this is possible and happens more frequently than we care to admit. Still, life goes on. There are safeguards, mostly in form of a legal and judicial system, to keep the occurrences of such instances low.

The cyber world is no different. In the absence of a better technology, we have to rely on cryptography and use it carefully.

However, cryptography by itself is quite inadequate for real life use. Exchange of encrypted files may work as means to share secret information in a small group of people that agree on the algorithm and a secret key or password beforehand, but is useless when the communicating parties may not know each other. Use of a digital signature as a means of proving authenticity requires that someone with appropriate authority should be able to substantiate the ownership claim of the private key. In cases where a private key is compromised, there has to be a way to invalidate the key and minimize the damage. Even transportation of keys requires defining a format so that software from different vendors can use them appropriately.

The solution to these and many other related problems lies in using agreed upon standards to store and communicate cryptographic information: conventions, policies and regulations for trust relationships and other related aspects of doing business. As we see in subsequent chapters, PKI standards, communication protocols like SSL and identification and authentication services define exactly such standards and conventions.

Legal Issues with Cryptography

The use of cryptography has traditionally been associated with military intelligence gathering and its use by criminals and terrorists has the potential to make law enforcement harder. Hence it should come as no surprise that governments tend to restrict its use. Other legal issues are patent related and arise due to the complex mathematical nature of the algorithms involved. Inventors of these algorithms tend to protect their intellectual property by patenting them and requiring that the user obtain a license.

All in all, the legal issues with cryptography fall into the following three categories:

1. **Export Control Issues.** The US government treats certain forms of cryptographic software and hardware as munitions and has placed them under export control. What it means is that a commercial entity seeking to export certain cryptographic libraries or other software using these libraries must obtain an export license first. In recent years, the export laws have eased somewhat and it has become possible to export freely a number of commercial grade cryptographic software packages. Most of the software and capabilities included in J2SE v1.4 falls under this category. However, it is possible to have a JCE provider with capabilities that warrant review by export control authorities and perhaps, an export license. A practical manifestation of this fact is that a vendor of JCE provider must get export clearance.
2. **Import Control Issues.** Somewhat less intuitive is the fact that certain countries restrict the use of certain types of cryptography within their jurisdiction. Under the jurisdiction of these countries, it is the responsibility of the user to ensure proper adherence to the law. J2SE v1.4 handles this by tying cryptographic capabilities to jurisdiction policy files. The jurisdiction files shipped with the J2SE v1.4 allow "strong" but "limited" cryptography by limiting the size of keys and other parameters. Those in the US must download and install separate policy files to be able to use "unlimited" capabilities.
3. **Patent Related Issues.** To avoid lawsuits related to patent infringement, it is recommended that you either use algorithms that are not patented, whose patents have expired, that are licensed for royalty free use or whose license you have obtained. The patent on RSA, the de-facto public key cryptography, was a big inhibitor for the wide spread use of public key cryptography before it expired in 2000. Algorithms available within J2SE v1.4 are either unencumbered from patent issues or are licensed royalty-free for use.

These are only broad guidelines that you must consider before deploying solutions using cryptographic components. Most of the time, it is the vendor of the security products who has to worry about these, but don't take chances. Extra care is required if you plan to use open source software freely available for download over the Internet, as you don't have the vendor to do the homework for legal compliance. When in doubt, consult legal counsel for proper guidance.

Notwithstanding, anything stated in this section or in the whole book, the author and publisher take no responsibility for any legal consequences resulting from following the advice offered or using any of the security techniques in this book. The laws regulating cryptography are complex, jurisdiction-dependent and keep changing all the time. It is your responsibility to ensure that you remain within the four walls of the law.

Summary

Java cryptographic services are defined independent of the underlying algorithms and implementations. This supports extensibility through the addition of newer algorithms in separate provider implementation without changing or adding the programmer visible classes. This extensibility is achieved through an architecture where the service engine classes expose the functionality, but hide the coupling with the implementation class. It also allows security capabilities to be extended by installing and configuring third-party providers. The same architecture is used by all security APIs, bringing a good deal of uniformity and ease of use.

Keys—secret key for symmetric encryption, and public-private key pairs for asymmetric encryption—are central to a number of cryptographic operations. Proper generation and handling of keys is essential for realizing the security offered by cryptography. Java Security API contains classes to handle keys as Java objects and has services to generate, store and load these keys. An important point to remember is that not all secret keys or public-private key pairs have the same structure or are generated by the same process—key pair used by RSA cannot be used by DSA and vice versa.

JCA and JCE contain the engine classes for basic cryptographic operations. Examples include `Signature` class for creating and verifying digital signature, `Cipher` class for symmetric and asymmetric encryption and decryption, `MessageDigest` class for computing and verifying message digest, `Mac` class for computing and verifying MAC and `KeyAgreement` class for key agreement operations. Encryption provides message confidentiality and digest helps in detecting changes to the message. MAC should be used in place of digest to prevent willful tampering when the complete message including the digest or MAC is exposed. Digital signature combines public key encryption with digest to provide non-repudiation.

You can perform these cryptographic operations using the command line utility `crypttool`. This allows experimentation with various combinations of services, algorithms and providers without any programming. You can also examine the source code of **`crypttool`** for sample code using the Java Security API.

Speed of cryptographic operations depends on the quality of implementation, algorithm used and the keysize. For J2SE v1.4 bundled providers, we found 56-bit DES encryption to be 2.5 times faster than 112-bit TripleDES encryption. For digital signature, we found RSA to be approximately 1.5 times faster than DSA for both signature creation and verification.

Cryptography requires standards and protocols to be useful in real life. Most of the applications require agreement about using cryptographic capabilities in a certain way. This is achieved through standards and protocols.

Further Reading

Most of the Java architecture for cryptography and API-related information presented in this chapter can be found in J2SE v1.4 specification and reference guides. Refer to these guides when in doubt. Authoritative documentation on Java classes and their methods can be found in javadocs of J2SE SDK. The book *Java Security* by Scott Oaks includes comprehensive information on security-related Java APIs and explains them with simple examples. This is a good book to have if you are developing security software and need to use cryptographic APIs directly.

Look at the book *Applied Cryptography* by Bruce Schneier for very detailed, almost encyclopedic, information on cryptographic operations, algorithms, protocols, attack vulnerabilities, performance and other related aspects such as patent and politico-legal issues. It's a must have if you plan to write your own provider and implement the cryptographic algorithms.

If you are interested in looking at working code as examples, dive into **`crypttool`** source code. It is quite modular and you will have no difficulty identifying relevant portions.