Java™ Cryptography Architecture

API Specification & Reference

Last Modified: 25 July 2004

Introduction
Design Principles
Architecture
Concepts

What's New in JCE in the Java 2 Platform Standard Edition 5

Core Classes and Interfaces
The Provider Class
How Provider Implementations are Requested and Supplied
Installing Providers
The Security Class
The MessageDigest Class
The Signature Class
Algorithm Parameters Classes
Algorithm Parameter Specification Interfaces and Classes
The AlgorithmParameterSpec Interface
The DSAParameterSpec Class
The AlgorithmParameters Class
The AlgorithmParameterGenerator Class
Key Interfaces
Key Specification Interfaces and Classes
The KeySpec Interface
The DSAPrivateKeySpec Class
The DSAPublicKeySpec Class
The RSAPrivateKeySpec Class
The RSAPrivateCrtKeySpec Class
The RSAMultiPrimePrivateCrtKeySpec Class
The RSAPublicKeySpec Class
The EncodedKeySpec Class
The PKCS8EncodedKeySpec Class
The X509EncodedKeySpec Class
The KeyFactory Class
The CertificateFactory Class
The KeyPair Class
The KeyPairGenerator Class
Key Management
Keystore Location
Keystore Implementation
The KeyStore Class
The SecureRandom Class
The Cipher Class
The CipherInputStream Class
The CipherOutputStream Class
Introduction

The Security API is a core API of the Java programming language, built around the `java.security` package (and its subpackages). This API is designed to allow developers to incorporate both low-level and high-level security functionality into their programs.

The first release of Security API in JDK 1.1 introduced the "Java Cryptography Architecture" (JCA), a framework for accessing and developing cryptographic functionality for the Java platform. In JDK 1.1, the JCA included APIs for digital signatures and message digests.

In subsequent releases, the Java 2 SDK significantly extended the Java Cryptography Architecture, as described in this document. It also upgraded the certificate management infrastructure to support X.509 v3 certificates, and introduced a new Java Security Architecture for fine-grain, highly configurable, flexible, and extensible access control.

The Java Cryptography Architecture encompasses the parts of the Java 2 SDK Security API related to cryptography, as well as a set of conventions and specifications provided in this document. It includes a "provider"
architecture that allows for multiple and interoperable cryptography implementations.

The Java™ Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects.

JCE was previously an optional package (extension) to the Java™ 2 SDK, Standard Edition (Java 2 SDK), versions 1.2.x and 1.3.x. JCE has been integrated into the Java 2 SDK since the 1.4 release.

The JCE API covers:

- Symmetric bulk encryption, such as DES, RC2, and IDEA
- Symmetric stream encryption, such as RC4
- Asymmetric encryption, such as RSA
- Password-based encryption (PBE)
- Key Agreement
- Message Authentication Codes (MAC)

J2SE 5 comes standard with a JCE provider named "SunJCE", which comes pre-installed and registered and which supplies the following cryptographic services:

- An implementation of the DES (FIPS PUB 46-1), Triple DES, and Blowfish encryption algorithms in the Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Propagating Cipher Block Chaining (PCBC) modes. (Note: Throughout this document, the terms "Triple DES" and "DES-EDE" will be used interchangeably.)
- Key generators for generating keys suitable for the DES, Triple DES, Blowfish, HMAC-MD5, and HMAC-SHA1 algorithms.
- An implementation of the MD5 with DES-CBC password-based encryption (PBE) algorithm defined in PKCS #5.
- "Secret-key factories" providing bi-directional conversions between opaque DES, Triple DES and PBE key objects and transparent representations of their underlying key material.
- An implementation of the Diffie-Hellman key agreement algorithm between two or more parties.
- A Diffie-Hellman key pair generator for generating a pair of public and private values suitable for the Diffie-Hellman algorithm.
- A Diffie-Hellman algorithm parameter generator.
- A Diffie-Hellman "key factory" providing bi-directional conversions between opaque Diffie-Hellman key objects and transparent representations of their underlying key material.
- Algorithm parameter managers for Diffie-Hellman, DES, Triple DES, Blowfish, and PBE parameters.
- An implementation of the HMAC-MD5 and HMAC-SHA1 keyed-hashing algorithms defined in RFC 2104.
- An implementation of the padding scheme described in PKCS #5.
- A keystore implementation for the proprietary keystore type named "JCEKS".

A Note on Terminology

The JCE within the JDK includes two software components:
• the framework that defines and supports cryptographic services that providers can supply implementations for. This framework includes everything in the javax.crypto package.

• a provider named "SunJCE"

Throughout this document, the term "JCE" by itself refers to the JCE framework in J2SE 5. Whenever the JCE provider supplied with J2SE 5 is mentioned, it will be referred to explicitly as the "SunJCE" provider.

Note: The most recent version of this JCA specification can be found online at: http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html.

Design Principles

The Java Cryptography Architecture (JCA) was designed around these principles:

• implementation independence and interoperability

• algorithm independence and extensibility

Implementation independence and algorithm independence are complementary; you can use cryptographic services, such as digital signatures and message digests, without worrying about the implementation details or even the algorithms that form the basis for these concepts. When complete algorithm-independence is not possible, the JCA provides standardized, algorithm-specific APIs. When implementation-independence is not desirable, the JCA lets developers indicate a specific implementation.

Algorithm independence is achieved by defining types of cryptographic "engines" (services), and defining classes that provide the functionality of these cryptographic engines. These classes are called engine classes, and examples are the MessageDigest, Signature, KeyFactory, and KeyPairGenerator classes.

Implementation independence is achieved using a "provider"-based architecture. The term Cryptographic Service Provider (used interchangeably with "provider" in this document) refers to a package or set of packages that implement one or more cryptographic services, such as digital signature algorithms, message digest algorithms, and key generation services. A program may simply request a particular type of object (such as a Signature object) implementing a particular service (such as the DSA signature algorithm) and get an implementation from one of the installed providers. If desired, a program may instead request an implementation from a specific provider. Providers may be updated transparently to the application, for example when faster or more secure versions are available.

Implementation interoperability means that various implementations can work with each other, use each other's keys, or verify each other's signatures. This would mean, for example, that for the same algorithms, a key generated by one provider would be usable by another, and a signature generated by one provider would be verifiable by another.

Algorithm extensibility means that new algorithms that fit in one of the supported engine classes can be added easily.

Architecture

Cryptographic Service Providers

The Java Cryptography Architecture introduced the notion of a Cryptographic Service Provider (used interchangeably with "provider" in this document). This term refers to a package (or a set of packages) that supplies a concrete implementation of a subset of the cryptography aspects of the Security API.

For example, in JDK 1.1 a provider could contain an implementation of one or more digital signature algorithms, message digest algorithms, and key generation algorithms. Java 2 SDK adds five additional types of services: key factories, keystore creation and management, algorithm parameter management, algorithm parameter generation, and certificate factories. It also enables a provider to supply a random number generation (RNG) algorithm.
Previously, RNGs were not provider-based; a particular algorithm was hard-coded in the JDK.

As previously noted, a program may simply request a particular type of object (such as a `Signature` object) for a particular service (such as the DSA signature algorithm) and get an implementation from one of the installed providers. Alternatively, the program can request the objects from a specific provider. (Each provider has a name used to refer to it.)

Sun's version of the Java runtime environment comes standard with a default provider, named SUN. Other Java runtime environments may not necessarily supply the SUN provider. The SUN provider package includes:

- An implementation of the Digital Signature Algorithm (DSA), described in NIST FIPS 186.
- An implementation of the MD5 (RFC 1321) and SHA-1 (NIST FIPS 180-1) message digest algorithms.
- A DSA key pair generator for generating a pair of public and private keys suitable for the DSA algorithm.
- A DSA algorithm parameter generator.
- A DSA algorithm parameter manager.
- A DSA key factory providing bi-directional conversions between (opaque) DSA private and public key objects and their underlying key material.
- An implementation of the proprietary "SHA1PRNG" pseudo-random number generation algorithm, following the recommendations in the IEEE P1363 standard (Appendix G.7).
- A certificate path builder and validator for PKIX, as defined in the Internet X.509 Public Key Infrastructure Certificate and CRL Profile (available as a draft from Internet Engineering Task Force at the time of this writing.).
- A certificate store implementation for retrieving certificates and CRLs from Collection and LDAP directories, using the PKIX LDAP V2 Schema (RFC 2587).
- A certificate factory for X.509 certificates and Certificate Revocation Lists (CRLs).
- A keystore implementation for the proprietary keystore type named JKS.

Each SDK installation has one or more provider packages installed. New providers may be added statically or dynamically (see the Provider and Security classes). The Java Cryptography Architecture offers a set of APIs that allow users to query which providers are installed and what services they support.

Clients may configure their runtime with different providers, and specify a preference order for each of them. The preference order is the order in which providers are searched for requested services when no specific provider is requested.

**Key Management**

A database called a "keystore" can be used to manage a repository of keys and certificates. A keystore is available to applications that need it for authentication or signing purposes.

Applications can access a keystore via an implementation of the KeyStore class, which is in the java.security package. A default KeyStore implementation is provided by Sun Microsystems. It implements the keystore as a file, using a proprietary keystore type (format) named "JKS".

Applications can choose different types of keystore implementations from different providers, using the getInstance factory method supplied in the KeyStore class.

See the Key Management section for more information.

**Concepts**

This section covers the major concepts introduced in the API.
Engine Classes and Algorithms

An *engine class* defines a cryptographic service in an abstract fashion (without a concrete implementation).

A cryptographic service is always associated with a particular algorithm or type, and it either provides cryptographic operations (like those for digital signatures or message digests), generates or supplies the cryptographic material (keys or parameters) required for cryptographic operations, or generates data objects (keystores or certificates) that encapsulate cryptographic keys (which can be used in a cryptographic operation) in a secure fashion. For example, two of the engine classes are the `Signature` and `KeyFactory` classes. The `Signature` class provides access to the functionality of a digital signature algorithm. A DSA `KeyFactory` supplies a DSA private or public key (from its encoding or transparent specification) in a format usable by the `initSign` or `initVerify` methods, respectively, of a DSA `Signature` object.

The Java Cryptography Architecture encompasses the classes of the Java 2 SDK Security package related to cryptography, including the engine classes. Users of the API request and use instances of the engine classes to carry out corresponding operations. The following engine classes are defined in Java 2 SDK:

- **MessageDigest**: used to calculate the message digest (hash) of specified data.
- **Signature**: used to sign data and verify digital signatures.
- **KeyPairGenerator**: used to generate a pair of public and private keys suitable for a specified algorithm.
- **KeyFactory**: used to convert opaque cryptographic keys of type `Key` into *key specifications* (transparent representations of the underlying key material), and vice versa.
- **CertificateFactory**: used to create public key certificates and Certificate Revocation Lists (CRLs).
- **KeyStore**: used to create and manage a *keystore*. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities.
- **AlgorithmParameters**: used to manage the parameters for a particular algorithm, including parameter encoding and decoding.
- **AlgorithmParameterGenerator**: used to generate a set of parameters suitable for a specified algorithm.
- **SecureRandom**: used to generate random or pseudo-random numbers.

In the 1.4 release of the Java 2 SDK, the following new engines were added:

- **CertPathBuilder**: used to build certificate chains (also known as certification paths).
- **CertPathValidator**: used to validate certificate chains.
- **CertStore**: used to retrieve *Certificates* and *CRLs* from a repository.

**Note:** A *generator* creates objects with brand-new contents, whereas a *factory* creates objects from existing material (for example, an encoding).

An engine class provides the interface to the functionality of a specific type of cryptographic service (independent of a particular cryptographic algorithm). It defines Application Programming Interface (API) methods that allow applications to access the specific type of cryptographic service it provides. The actual implementations (from one or more providers) are those for specific algorithms. The `Signature` engine class, for example, provides access to the functionality of a digital signature algorithm. The actual implementation supplied in a `SignatureSpi` subclass would be that for a specific kind of signature algorithm, such as SHA-1 with DSA, SHA-1 with RSA, or MD5 with RSA.

The application interfaces supplied by an engine class are implemented in terms of a Service Provider Interface (SPI). That is, for each engine class, there is a corresponding abstract SPI class, which defines the SPI methods that cryptographic service providers must implement.
An instance of an engine class, the API object, encapsulates (as a private field) an instance of the corresponding SPI class, the SPI object. All API methods of an API object are declared final and their implementations invoke the corresponding SPI methods of the encapsulated SPI object. An instance of an engine class (and of its corresponding SPI class) is created by a call to the getInstance factory method of the engine class.

The name of each SPI class is the same as that of the corresponding engine class, followed by Spi. For example, the SPI class corresponding to the Signature engine class is the SignatureSpi class.

Each SPI class is abstract. To supply the implementation of a particular type of service, for a specific algorithm, a provider must subclass the corresponding SPI class and provide implementations for all the abstract methods.

Another example of an engine class is the MessageDigest class, which provides access to a message digest algorithm. Its implementations, in MessageDigestSpi subclasses, may be those of various message digest algorithms such as SHA-1, MD5, or MD2.

As a final example, the KeyFactory engine class supports the conversion from opaque keys to transparent key specifications, and vice versa. (See the Key Specification Interfaces and Classes section.) The KeyFactorySpi subclass supplies an actual implementation for a specific type of keys, for example, DSA public and private keys.

Implementations and Providers

Implementations for various cryptographic services are provided by JCA Cryptographic Service Providers. Cryptographic service providers are essentially packages that supply one or more cryptographic service implementations. The Engine Classes and Algorithms section includes a list of implementations supplied by SUN, the Java 2 SDK's default provider.

Other providers may define their own implementations of these services or of other services, such as one of the RSA-based signature algorithms or the MD2 message digest algorithm.

Factory Methods to Obtain Implementation Instances

For each engine class in the API, a particular implementation is requested and instantiated by calling a factory method on the engine class. A factory method is a static method that returns an instance of a class.

The basic mechanism for obtaining an appropriate Signature object, for example, is as follows: A user requests such an object by calling the getInstance method in the Signature class, specifying the name of a signature algorithm (such as "SHA1withDSA"), and, optionally, the name of the provider or the Provider class. The getInstance method finds an implementation that satisfies the supplied algorithm and provider parameters. If no provider is specified, getInstance searches the registered providers, in preference order, for one with an implementation of the specified algorithm. See The Provider Class for more information about registering providers.

Cryptographic Concepts

This section provides a high-level description of the concepts implemented by the API, and the exact meaning of the technical terms used in the API specification.

Encryption and Decryption

Encryption is the process of taking data (called cleartext) and a short string (a key), and producing data (ciphertext) meaningless to a third-party who does not know the key. Decryption is the inverse process: that of taking ciphertext and a short key string, and producing cleartext.

Password-Based Encryption

Password-Based Encryption (PBE) derives an encryption key from a password. In order to make the task of getting from password to key very time-consuming for an attacker, most PBE implementations will mix in a random number, known as a salt, to create the key.
Cipher

Encryption and decryption are done using a cipher. A cipher is an object capable of carrying out encryption and decryption according to an encryption scheme (algorithm).

Key Agreement

Key agreement is a protocol by which 2 or more parties can establish the same cryptographic keys, without having to exchange any secret information.

Message Authentication Code

A Message Authentication Code (MAC) provides a way to check the integrity of information transmitted over or stored in an unreliable medium, based on a secret key. Typically, message authentication codes are used between two parties that share a secret key in order to validate information transmitted between these parties.

A MAC mechanism that is based on cryptographic hash functions is referred to as HMAC. HMAC can be used with any cryptographic hash function, e.g., MD5 or SHA-1, in combination with a secret shared key. HMAC is specified in RFC 2104.

What's New in JCE in J2SE 5

Here are the differences in JCE between v1.4 and J2SE 5:

- Support for Additional Features of PKCS #11
- Integration with Solaris Cryptographic Framework
- Support for ECC Algorithm
- Added ByteBuffer API Support to JCA/JCE
- Support for RC2ParameterSpec
- Full support for XML Encryption RSA-OAEP Algorithm
- Simplified retrieval of PKCS8EncodedKeySpec fromjavax.crypto.EncryptedPrivateKeyInfo
- Support for "PBEWithSHA1AndDESede" and "PBEWithSHA1AndRC2_40" Ciphers
- Support for XML Encryption Padding Algorithm in JCE Block Encryption Ciphers
- Ability to Dynamically Determine Maximum Allowable Key Length
- Support for RSA encryption to SunJCE provider
- Support for RC2 and ARCFOUR Ciphers to SunJCE provider
- Support for HmacSHA256, HmacSHA384 and HmacSHA512

Support for PKCS #11 Based Crypto Provider

In J2SE 5, a JCA/JCE provider, SunPKCS11 that acts as a generic gateway to the native PKCS#11 API has been implemented. PKCS#11 is the de-facto standard for crypto accelerators and also widely used to access cryptographic smartcards. The administrator/user can configure this provider to talk any PKCS#11 v2.x compliant token.

Here's an example of the configuration file format.
Integration with Solaris Cryptographic Framework

On Solaris 10, the default Java security provider configuration has been changed in J2SE 5 to include an instance of the SunPKCS11 provider that uses the Solaris Cryptographic Framework. It is the provider with the highest precedence thereby allowing all existing applications to take advantage of the improved performance on Solaris 10. There is no change in behavior on Solaris 8 and Solaris 9 systems.

As a result of this change, many cryptographic operations will execute several times as fast as before on all Solaris 10 systems. On systems with cryptographic hardware acceleration, the performance improvements may be two orders of magnitude.

Support for ECC Algorithm

Prior to J2SE 5, the JCA/JCE framework did not include support classes for ECC-related crypto algorithms. Users who wanted to use ECC had to depend on a 3rd party library that implemented ECC. However, this did not integrate well with existing JCA/JCE framework.

Starting in J2SE 5, full support for ECC classes to facilitate providers that support ECC have been included.

The following interfaces have been added:

- java.security.spec.ECField
- java.security.interfaces.ECKey
- java.security.interfaces.ECPublicKey
- java.security.interfaces.ECPrivateKey

The following classes have been added:

- java.security.spec.ECFieldF2m
- java.security.spec.ECFieldFp
- java.security.spec.ECGenParameterSpec
- java.security.spec.ECParameterSpec
- java.security.spec.ECPoint
- java.security.spec.ECPrivateKeySpec
- java.security.spec.ECPublicKeySpec

Added ByteBuffer API Support

Methods that take ByteBuffer arguments have been added to the JCE API and SPI classes that are used to process bulk data. Providers can override the engine* methods if they can process ByteBuffers more efficiently than byte[].

The following JCE methods have been added to support ByteBuffers:

- javax.crypto.Mac.update(ByteBuffer input)
- javax.crypto.MacSpi.engineUpdate(ByteBuffer input)
- javax.crypto.Cipher.update(ByteBuffer input, ByteBuffer output)
- javax.crypto.Cipher.doFinal(ByteBuffer input, ByteBuffer output)
- javax.crypto.CipherSpi.engineUpdate(ByteBuffer input, ByteBuffer output)
- javax.crypto.CipherSpi.engineDoFinal(ByteBuffer input, ByteBuffer output)

The following JCA methods have been added to support ByteBuffers:

- java.security.MessageDigest.update(ByteBuffer input)
- java.security.Signature.update(ByteBuffer data)
- java.security.SignatureSpi.engineUpdate(ByteBuffer data)
- java.security.MessageDigestSpi.engineUpdate(ByteBuffer input)

Support for RC2ParameterSpec

The RC2 algorithm implementation has been enhanced in J2SE 5 to support effective key size that is
distinct from the length of the input key.

**Full support for XML Encryption RSA-OAEP Algorithm**

Prior to J2SE 5, JCE did not define any parameter class for specifying the non-default values used in OAEP and PSS padding as defined in PKCS#1 v2.1 and the RSA-OAEP Key Transport algorithm in the [W3C Recommendation for XML Encryption](https://www.w3.org/TR/2002/REC-xmlenc1-20020815). Therefore, there was no generic way for applications to specify non-default values used in OAEP and PSS padding.

In J2SE 5, new parameter classes have been added to fully support OAEP padding and the existing PSS parameter class was enhanced with APIs to fully support RSA PSS signature implementations. Also, SunJCE provider has been enhanced to accept `OAEPParameterSpec` when OAEPPadding is used.

The following classes have been added:

- `javax.crypto.spec.OAEPParameterSpec`
- `javax.crypto.spec.PSource`
- `javax.security.spec.MGF1ParameterSpec`

The following methods and fields have been added to `java.security.spec.PSSParameterSpec`:

```java
public static final PSSParameterSpec DEFAULT
public PSSParameterSpec(String mdName, String mgfName,
                        AlgorithmParameterSpec mgfSpec,
                        int saltLen, int trailerField)
public String getDigestAlgorithm()
public String getMGFAlgorithm()
public AlgorithmParameterSpec getMGFParameters()
public int getTrailerField()
```

**Simplified Retrieval of PKCS8EncodedKeySpec from javax.crypto.EncryptedPrivateKeyInfo**

In J2SE 5, `javax.crypto.EncryptedPrivateKeyInfo` only has one method, `getKeySpec(Cipher)` for retrieving the `PKCS8EncodedKeySpec` from the encrypted data. This limitation requires users to specify a cipher which is initialized with the decryption key and parameters. When users only have the decryption key, they would have to first retrieve the parameters out of this `EncryptedPrivateKeyInfo` object, get hold of matching `Cipher` implementation, initialize it, and then call the `getKeySpec(Cipher)` method.

To make `EncryptedPrivateKeyInfo` easier to use and to make its API consistent with `javax.crypto.SealedObject`, the following methods have been added to `javax.crypto.EncryptedPrivateKeyInfo`:

```java
getKeySpec(Key decryptKey)
getKeySpec(Key decryptKey, String provider)
```

**Ability to Dynamically Determine Maximum Allowable Key Length**

In 1.4.2, the crypto jurisdiction policy files bundled in J2SE limits the maximum key length (and parameter value for some crypto algorithms) that can be used for encryption/decryption. Users who desire unlimited version of crypto jurisdiction files must download them separately.

Also, an exception is thrown when the Cipher instance is initialized with keys (or parameters for certain crypto algorithms) exceeds the maximum values allowed by the crypto jurisdiction files.

In J2SE 5, the `Cipher` class has been updated to provide the maximum values for key length and parameters configured in the jurisdiction policy files, so that applications can use a shorter key length when the default (limited strength) jurisdiction policy files are installed.

The following methods have been added to `javax.crypto.Cipher`:

```java
public static final int getMaxAllowedKeyLength(String transformation)
    throws NoSuchAlgorithmException
public static final AlgorithmParameterSpec
    getMaxAllowedParameterSpec(String transformation)
    throws NoSuchAlgorithmException;
```
Support for HmacSHA256, HmacSHA384, HmacSHA512

Support for HmacSHA-256, HmacSHA-384, and HmacSHA-512 algorithms have been added to J2SE 5.

Support for RSA Encryption to SunJCE Provider

A publicly accessible RSA encryption implementation has been added to the SunJCE provider.

Support for RC2 and ARCFOUR Ciphers to SunJCE Provider

The SunJCE provider now implements the RC2 (RFC 2268) and ARCFOUR (an RC4™-compatible algorithm) ciphers.

Support for "PBEWithSHA1AndDESede" and "PBEWithSHA1AndRC2_40" Ciphers

Added support for PBEWithSHA1AndDESede and PBEWithSHA1AndRC2_40 ciphers in SunJCE provider.

Support for XML Encryption Padding Algorithm in JCE Block Encryption Ciphers


To allow Sun's provider to be used by XML Encryption implementations and JSR 106 providers, we have added support for this padding in J2SE 5.

Core Classes and Interfaces

This section discusses the core classes and interfaces provided in the Java Cryptography Architecture:

- the Provider and Security classes
- the MessageDigest, Signature, KeyPairGenerator, KeyFactory, AlgorithmParameters, AlgorithmParameterGenerator, CertificateFactory, KeyStore, SecureRandom, CertPathBuilder, CertPathValidator, and CertStore.

engine classes
- the Key interfaces and classes
- the Algorithm Parameter Specification Interfaces and Classes and the Key Specification Interfaces and Classes

This section shows the signatures of the main methods in each class and interface. Examples for some of these classes (MessageDigest, Signature, KeyPairGenerator, SecureRandom, KeyFactory, and key specification classes) are supplied in the corresponding Examples sections. The complete reference documentation for the relevant Security API packages can be found in:

- java.security package summary
- java.security.spec package summary
- java.security.interfaces package summary
- java.security.cert package summary

The Provider Class

The term "Cryptographic Service Provider" (used interchangeably with "provider" in this document) refers to a package or set of packages that supply a concrete implementation of a subset of the Java 2 SDK Security API cryptography features. The Provider class is the interface to such a package or set of packages. It has methods for accessing the provider name, version number, and other information. Please note that in addition to registering implementations of cryptographic services, the Provider class can also be used to register implementations of
other security services that might get defined as part of the Java 2 SDK Security API or one of its extensions.

To supply implementations of cryptographic services, an entity (e.g., a development group) writes the implementation code and creates a subclass of the Provider class. The constructor of the Provider subclass sets the values of various properties; the Java 2 SDK Security API uses these values to look up the services that the provider implements. In other words, the subclass specifies the names of the classes implementing the services.

There are several types of services that can be implemented by provider packages; for more information, see Engine Classes and Algorithms.

The different implementations may have different characteristics. Some may be software-based, while others may be hardware-based. Some may be platform-independent, while others may be platform-specific. Some provider source code may be available for review and evaluation, while some may not. The Java Cryptography Architecture (JCA) lets both end-users and developers decide what their needs are.

In this section we explain how end-users install the cryptography implementations that fit their needs, and how developers request the implementations that fit theirs.

---

**Note:** For information about implementing a provider, see the guide How To Implement a Provider for the Java Cryptography Architecture.

---

### How Provider Implementations Are Requested and Supplied

For each engine class in the API, a particular implementation is requested and instantiated by calling a getInstance method on the engine class, specifying the name of the desired algorithm and, optionally, the name of the provider (or the Provider class) whose implementation is desired.

If no provider is specified, getInstance searches the registered providers for an implementation of the requested cryptographic service associated with the named algorithm. In any given Java Virtual Machine (JVM), providers are installed in a given preference order, the order in which the provider list is searched if a specific provider is not requested. For example, suppose there are two providers installed in a JVM, PROVIDER_1 and PROVIDER_2. Assume that:

- PROVIDER_1 implements SHA1withDSA, SHA-1, MD5, DES, and DES3.
  - PROVIDER_1 has preference order 1 (the highest priority).
- PROVIDER_2 implements SHA1withDSA, MD5withRSA, MD2withRSA, MD2, MD5, RC4, RC5, DES, and RSA.
  - PROVIDER_2 has preference order 2.

Now let's look at three scenarios:

1. If we are looking for an MD5 implementation. Both providers supply such an implementation. The PROVIDER_1 implementation is returned since PROVIDER_1 has the highest priority and is searched first.
2. If we are looking for an MD5withRSA signature algorithm, PROVIDER_1 is first searched for it. No implementation is found, so PROVIDER_2 is searched. Since an implementation is found, it is returned.
3. Suppose we are looking for a SHA1withRSA signature algorithm. Since no installed provider implements it, a NoSuchAlgorithmException is thrown.

The getInstance methods that include a provider argument are for developers who want to specify which provider they want an algorithm from. A federal agency, for example, will want to use a provider implementation that has received federal certification. Let's assume that the SHA1withDSA implementation from PROVIDER_1 has not received such certification, while the DSA implementation of PROVIDER_2 has received it.

A federal agency program would then have the following call, specifying PROVIDER_2 since it has the certified implementation:

```java
Signature dsa = Signature.getInstance("SHA1withDSA", "PROVIDER_2");
```
In this case, if PROVIDER_2 was not installed, a NoSuchProviderException would be thrown, even if another installed provider implements the algorithm requested.

A program also has the option of getting a list of all the installed providers (using the getProviders method in the Security class) and choosing one from the list.

Installing Providers

There are two parts to installing a provider: installing the provider package classes, and configuring the provider.

Installing the Provider Classes

There are two possible ways to install the provider classes:

1. Place a zip or JAR file containing the classes anywhere in your classpath.
2. Supply your provider JAR file as an "installed" or "bundled" extension. For more information on how to deploy an extension, see How is an extension deployed?

Configuring the Provider

The next step is to add the provider to your list of approved providers. This step can be done statically by editing the java.security file in the lib/security directory of the SDK; therefore, if the SDK is installed in a directory called j2sdk1.2, the file would be j2sdk1.2/lib/security/java.security. One of the types of properties you can set in java.security has the following form:

```
security.provider.n=masterClassName
```

This declares a provider, and specifies its preference order n. The preference order is the order in which providers are searched for requested algorithms (when no specific provider is requested). The order is 1-based: 1 is the most preferred, followed by 2, and so on.

The exact format of the masterClassName must specify the provider's master class. The provider's documentation will specify its master class. This class is always a subclass of the Provider class. The subclass constructor sets the values of various properties that are required for the Java Cryptography API to look up the algorithms or other facilities the provider implements.

Suppose that the master class is COM.acme.provider.Acme, and that you would like to configure Acme as your third preferred provider. To do so, you would add the following line to the java.security file:

```
security.provider.3=COM.acme.provider.Acme
```

Providers may also be registered dynamically. To do so, call either the addProvider or insertProviderAt method in the Security class. This type of registration is not persistent and can only be done by "trusted" programs. See Security.

Provider Class Methods

Each Provider class instance has a (currently case-sensitive) name, a version number, and a string description of the provider and its services. You can query the Provider instance for this information by calling the following methods:

```
public String getName()
public double getVersion()
public String getInfo()
```

The Security Class

The Security class manages installed providers and security-wide properties. It only contains static methods and
is never instantiated. The methods for adding or removing providers, and for setting Security properties, can only be executed by a trusted program. Currently, a "trusted program" is either

- a local application not running under a security manager, or
- an applet or application with permission to execute the specified method (see below).

The determination that code is considered trusted to perform an attempted action (such as adding a provider) requires that the applet is granted permission for that particular action.

For example, in the Policy reference implementation, the policy configuration file(s) for a SDK installation specify what permissions (which types of system resource accesses) are allowed by code from specified code sources. (See below and the "Default Policy Implementation and Policy File Syntax" and "Java Security Architecture Specification" files for more information.)

Code being executed is always considered to come from a particular "code source". The code source includes not only the location (URL) where the applet originated from, but also a reference to the public key(s) corresponding to the private key(s) used to sign the code. Public keys in a code source are referenced by (symbolic) alias names from the user's keystore.

In a policy configuration file, a code source is represented by two components: a code base (URL), and an alias name (preceded by signedBy), where the alias name identifies the keystore entry containing the public key that must be used to verify the code's signature.

Each "grant" statement in such a file grants a specified code source a set of permissions, specifying which actions are allowed.

Here is a sample policy configuration file:

```java
grant codeBase "file:/home/sysadmin/", signedBy "sysadmin" {
    permission java.security.SecurityPermission "insertProvider.*";
    permission java.security.SecurityPermission "removeProvider.*";
    permission java.security.SecurityPermission "putProviderProperty.*";
}
```

This configuration file specifies that only code loaded from a signed JAR file from beneath the /home/sysadmin/ directory on the local file system can add or remove providers or set provider properties. (Note that the signature of the JAR file can be verified using the public key referenced by the alias name sysadmin in the user's keystore.)

Either component of the code source (or both) may be missing. Here's an example of a configuration file where codeBase is missing:

```java
grant signedBy "sysadmin" {
    permission java.security.SecurityPermission "insertProvider.*";
    permission java.security.SecurityPermission "removeProvider.*";
}
```

If this policy is in effect, code that comes in a JAR File signed by sysadmin can add/remove providers--regardless of where the JAR File originated.

Here's an example without a signer:

```java
grant codeBase "file:/home/sysadmin/" {
    permission java.security.SecurityPermission "insertProvider.*";
    permission java.security.SecurityPermission "removeProvider.*";
}
```

In this case, code that comes from anywhere within the /home/sysadmin/ directory on the local filesystem can add/remove providers. The code does not need to be signed.

An example where neither codeBase nor signedBy is included is:

```java
grant {
    permission java.security.SecurityPermission "insertProvider.*";
    permission java.security.SecurityPermission "removeProvider.*";
}
```
Here, with both code source components missing, any code (regardless of where it originates, or whether or not it is signed, or who signed it) can add/remove providers.

Managing Providers

The following tables summarize the methods in the Security class you can use to query which Providers are installed, as well as to install or remove providers at runtime.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quering Providers</strong></td>
<td></td>
</tr>
<tr>
<td>static Provider[] getProviders()</td>
<td>Returns an array containing all the installed providers (technically, the Provider subclass for each package provider). The order of the Providers in the array is their preference order.</td>
</tr>
<tr>
<td>static Provider getProvider(String providerName)</td>
<td>Returns the Provider named providerName. It returns null if the Provider is not found.</td>
</tr>
<tr>
<td><strong>Adding Providers</strong></td>
<td></td>
</tr>
<tr>
<td>static int addProvider(Provider provider)</td>
<td>Adds a Provider to the end of the list of installed Providers. It returns the preference position in which the Provider was added, or -1 if the Provider was not added because it was already installed.</td>
</tr>
<tr>
<td>static int insertProviderAt(Provider provider, int position)</td>
<td>Adds a new Provider at a specified position. If the given provider is installed at the requested position, the provider formerly at that position and all providers with a position greater than position are shifted up one position (towards the end of the list). This method returns the preference position in which the Provider was added, or -1 if the Provider was not added because it was already installed.</td>
</tr>
<tr>
<td><strong>Removing Providers</strong></td>
<td></td>
</tr>
<tr>
<td>static void removeProvider(String name)</td>
<td>Removes the Provider with the specified name. It returns silently if the provider is not installed. When the specified provider is removed, all providers located at a position greater than where the specified provider was are shifted down one position (towards the head of the list of installed providers).</td>
</tr>
</tbody>
</table>

Note: If you want to change the preference position of a provider, you must first remove it, and then insert it back in at the new preference position.

Security Properties

The Security class maintains a list of system-wide security properties. These properties are accessible and settable by a trusted program via the following methods:

```java
static String getProperty(String key)
static void setProperty(String key, String datum)
```

The MessageDigest Class

The MessageDigest class is an engine class designed to provide the functionality of cryptographically secure message digests such as SHA-1 or MD5. A cryptographically secure message digest takes arbitrary-sized input (a byte array), and generates a fixed-size output, called a digest or hash. A digest has two properties:
It should be computationally infeasible to find two messages that hashed to the same value.

The digest should not reveal anything about the input that was used to generate it.

Message digests are used to produce unique and reliable identifiers of data. They are sometimes called the "digital fingerprints" of data.

Creating a MessageDigest Object

The first step for computing a digest is to create a message digest instance. As with all engine classes, the way to get a MessageDigest object for a particular type of message digest algorithm is to call the getInstance static factory method on the MessageDigest class:

```java
static MessageDigest getInstance(String algorithm)
```

**Note:** The algorithm name is not case-sensitive. For example, all the following calls are equivalent:

```java
MessageDigest.getInstance("SHA-1")
MessageDigest.getInstance("sha-1")
MessageDigest.getInstance("sHa-1")
```

A caller may optionally specify the name of a provider or a Provider instance, which guarantees that the implementation of the algorithm requested is from the specified provider:

```java
static MessageDigest getInstance(String algorithm, String provider)
static MessageDigest getInstance(String algorithm, Provider provider)
```

A call to getInstance returns an initialized message digest object. It thus does not need further initialization.

Updating a Message Digest Object

The next step for calculating the digest of some data is to supply the data to the initialized message digest object. This is done by calling one of the update methods:

```java
void update(byte input)
void update(byte[] input)
void update(byte[] input, int offset, int len)
```

Computing the Digest

After the data has been supplied by calls to update methods, the digest is computed using a call to one of the digest methods:

```java
byte[] digest()
byte[] digest(byte[] input)
int digest(byte[] buf, int offset, int len)
```

The first two methods return the computed digest. The latter method stores the computed digest in the provided buffer buf, starting at offset. len is the number of bytes in buf allotted for the digest. The method returns the number of bytes actually stored in buf.

A call to the digest method that takes an input byte array argument is equivalent to making a call to

```java
void update(byte[] input)
```

with the specified input, followed by a call to the digest method without any arguments.

Please see the Examples section for more details.

The Signature Class
The `Signature` class is an engine class designed to provide the functionality of a cryptographic digital signature algorithm such as DSA or RSA with MD5. A cryptographically secure signature algorithm takes arbitrary-sized input and a private key and generates a relatively short (often fixed-size) string of bytes, called the signature, with the following properties:

- Given the public key corresponding to the private key used to generate the signature, it should be possible to verify the authenticity and integrity of the input.
- The signature and the public key do not reveal anything about the private key.

A `Signature` object can be used to sign data. It can also be used to verify whether or not an alleged signature is in fact the authentic signature of the data associated with it. Please see the Examples section for an example of signing and verifying data.

**Signature Object States**

`Signature` objects are modal objects. This means that a `Signature` object is always in a given state, where it may only do one type of operation. States are represented as final integer constants defined in their respective classes.

The three states a `Signature` object may have are:

- `UNINITIALIZED`
- `SIGN`
- `VERIFY`

When it is first created, a `Signature` object is in the `UNINITIALIZED` state. The `Signature` class defines two initialization methods, `initSign` and `initVerify`, which change the state to `SIGN` and `VERIFY`, respectively.

**Creating a Signature Object**

The first step for signing or verifying a signature is to create a `Signature` instance. As with all engine classes, the way to get a `Signature` object for a particular type of signature algorithm is to call the `getInstance` static factory method on the `Signature` class:

```java
static Signature getInstance(String algorithm)
```

*Note: The algorithm name is not case-sensitive.*

A caller may optionally specify the name of a provider or the `Provider` class, which will guarantee that the implementation of the algorithm requested is from the named provider:

```java
static Signature getInstance(String algorithm, String provider)
static Signature getInstance(String algorithm, Provider provider)
```

**Initializing a Signature Object**

A `Signature` object must be initialized before it is used. The initialization method depends on whether the object is going to be used for signing or for verification.

If it is going to be used for signing, the object must first be initialized with the private key of the entity whose signature is going to be generated. This initialization is done by calling the method:

```java
final void initSign(PrivateKey privateKey)
```

This method puts the `Signature` object in the `SIGN` state.

If instead the `Signature` object is going to be used for verification, it must first be initialized with the public key of the entity whose signature is going to be verified. This initialization is done by calling...
either of these methods:

```java
final void initVerify(PublicKey publicKey)
final void initVerify(Certificate certificate)
```

This method puts the `Signature` object in the `VERIFY` state.

**Signing**

If the `Signature` object has been initialized for signing (if it is in the `SIGN` state), the data to be signed can then be supplied to the object. This is done by making one or more calls to one of the `update` methods:

```java
final void update(byte b)
final void update(byte[] data)
final void update(byte[] data, int off, int len)
```

Calls to the `update` method(s) should be made until all the data to be signed has been supplied to the `Signature` object.

To generate the signature, simply call one of the `sign` methods:

```java
final byte[] sign()
final int sign(byte[] outbuf, int offset, int len)
```

The first method returns the signature result in a byte array. The second stores the signature result in the provided buffer `outbuf`, starting at `offset`. `len` is the number of bytes in `outbuf` allotted for the signature. The method returns the number of bytes actually stored.

Signature encoding is algorithm specific. See Appendix B for more information about the use of ASN.1 encoding in the Java Cryptography Architecture.

A call to a `sign` method resets the signature object to the state it was in when previously initialized for signing via a call to `initSign`. That is, the object is reset and available to generate another signature with the same private key, if desired, via new calls to `update` and `sign`.

Alternatively, a new call can be made to `initSign` specifying a different private key, or to `initVerify` (to initialize the `Signature` object to verify a signature).

**Verifying**

If the `Signature` object has been initialized for verification (if it is in the `VERIFY` state), it can then verify if an alleged signature is in fact the authentic signature of the data associated with it. To start the process, the data to be verified (as opposed to the signature itself) is supplied to the object. The data is passed to the object by calling one of the `update` methods:

```java
final void update(byte b)
final void update(byte[] data)
final void update(byte[] data, int off, int len)
```

Calls to the `update` method(s) should be made until all the data to be verified has been supplied to the `Signature` object. The signature can now be verified by calling one of the `verify` methods:

```java
final boolean verify(byte[] signature)
final boolean verify(byte[] signature, int offset, int length)
```

The argument must be a byte array containing the signature. The argument must be a byte array containing the signature. This byte array would hold the signature bytes which were returned by a previous call to one of the `sign` methods.

The `verify` method returns a `boolean` indicating whether or not the encoded signature is the authentic signature of the data supplied to the `update` method(s).
A call to the `verify` method resets the signature object to its state when it was initialized for verification via a call to `initVerify`. That is, the object is reset and available to verify another signature from the identity whose public key was specified in the call to `initVerify`.

Alternatively, a new call can be made to `initVerify` specifying a different public key (to initialize the `Signature` object for verifying a signature from a different entity), or to `initSign` (to initialize the `Signature` object for generating a signature).

Algorithm Parameters Classes

Algorithm Parameter Specification Interfaces and Classes

An algorithm parameter specification is a transparent representation of the sets of parameters used with an algorithm.

A transparent representation of a set of parameters means that you can access each parameter value in the set individually. You can access these values through one of the `get` methods defined in the corresponding specification class (e.g., `DSAParameterSpec` defines `getP`, `getQ`, and `getG` methods, to access `p`, `q`, and `g`, respectively).

In contrast, the `AlgorithmParameters` class supplies an opaque representation, in which you have no direct access to the parameter fields. You can only get the name of the algorithm associated with the parameter set (via `getAlgorithm`) and some kind of encoding for the parameter set (via `getEncoded`).

The algorithm parameter specification interfaces and classes in the `java.security.spec` package are described in the following sections.

The `AlgorithmParameterSpec` Interface

`AlgorithmParameterSpec` is an interface to a transparent specification of cryptographic parameters.

This interface contains no methods or constants. Its only purpose is to group (and provide type safety for) all parameter specifications. All parameter specifications must implement this interface.

The `DSAParameterSpec` Class

This class (which implements the `AlgorithmParameterSpec` interface) specifies the set of parameters used with the DSA algorithm. It has the following methods:

```java
BigInteger getP()
BigInteger getQ()
BigInteger getG()
```

These methods return the DSA algorithm parameters: the prime `p`, the sub-prime `q`, and the base `g`.

The `AlgorithmParameters` Class

The `AlgorithmParameters` class is an engine class that provides an opaque representation of cryptographic parameters.

An opaque representation is one in which you have no direct access to the parameter fields; you can only get the name of the algorithm associated with the parameter set and some kind of encoding for the parameter set. This is in contrast to a transparent representation of parameters, in which you can access each value individually, through one of the `get` methods defined in the corresponding specification class. Note that you can call the `AlgorithmParameters.getParameterSpec` method to convert an `AlgorithmParameters` object to a transparent specification (see the following section).
Creating an AlgorithmParameters Object

As with all engine classes, the way to get an AlgorithmParameters object for a particular type of algorithm is to call the getInstance static factory method on the AlgorithmParameters class:

```
static AlgorithmParameters getInstance(String algorithm)
```

**Note:** The algorithm name is not case-sensitive.

A caller may optionally specify the name of a provider or the Provider class, which will guarantee that the algorithm parameter implementation requested is from the named provider:

```
static AlgorithmParameters getInstance(String algorithm, String provider)
static AlgorithmParameters getInstance(String algorithm, Provider provider)
```

Initializing an AlgorithmParameters Object

Once an AlgorithmParameters object is instantiated, it must be initialized via a call to init, using an appropriate parameter specification or parameter encoding:

```
void init(AlgorithmParameterSpec paramSpec)
void init(byte[] params)
void init(byte[] params, String format)
```

In these init methods, params is an array containing the encoded parameters, and format is the name of the decoding format. In the init method with a params argument but no format argument, the primary decoding format for parameters is used. The primary decoding format is ASN.1, if an ASN.1 specification for the parameters exists.

**Note:** AlgorithmParameters objects can be initialized only once. They are not reusable.

Obtaining the Encoded Parameters

A byte encoding of the parameters represented in an AlgorithmParameters object may be obtained via a call to getEncoded:

```
byte[] getEncoded()
```

This method returns the parameters in their primary encoding format. The primary encoding format for parameters is ASN.1, if an ASN.1 specification for this type of parameters exists.

If you want the parameters returned in a specified encoding format, use

```
byte[] getEncoded(String format)
```

If format is null, the primary encoding format for parameters is used, as in the other getEncoded method.

**Note:** In the default AlgorithmParameters implementation, supplied by the "SUN" provider, the format argument is currently ignored.

Converting an AlgorithmParameters Object to a Transparent Specification

A transparent parameter specification for the algorithm parameters may be obtained from an AlgorithmParameters object via a call to getParameterSpec:

```
```
AlgorithmParameterSpec getParameterSpec(Class paramSpec)

paramSpec identifies the specification class in which the parameters should be returned. The specification class could be, for example, DSAParameterSpec.class to indicate that the parameters should be returned in an instance of the DSAParameterSpec class. (This class is in the java.security.spec package.)

The AlgorithmParameterGenerator Class

The AlgorithmParameterGenerator class is an engine class used to generate a set of parameters suitable for a certain algorithm (the algorithm specified when an AlgorithmParameterGenerator instance is created).

Creating an AlgorithmParameterGenerator Object

As with all engine classes, the way to get an AlgorithmParameterGenerator object for a particular type of algorithm is to call the getInstance static factory method on the AlgorithmParameterGenerator class:

static AlgorithmParameterGenerator getInstance(
    String algorithm)

Note: The algorithm name is not case-sensitive.

A caller may optionally specify the name of a provider or the Provider class, which will guarantee that the algorithm parameter generator implementation is from the named provider:

static AlgorithmParameterGenerator getInstance(
    String algorithm,
    String provider)

static AlgorithmParameterGenerator getInstance(
    String algorithm,
    Provider provider)

Initializing an AlgorithmParameterGenerator Object

The AlgorithmParameterGenerator object can be initialized in two different ways: an algorithm-independent manner or an algorithm-specific manner.

The algorithm-independent approach uses the fact that all parameter generators share the concept of a "size" and a source of randomness. The measure of size is universally shared by all algorithm parameters, though it is interpreted differently for different algorithms. For example, in the case of parameters for the DSA algorithm, "size" corresponds to the size of the prime modulus, in bits. (See Appendix B: Algorithms for information about the sizes for specific algorithms.) When using this approach, algorithm-specific parameter generation values--if any--default to some standard values. One init method that takes these two universally shared types of arguments:

void init(int size, SecureRandom random);

Another init method takes only a size argument and uses a system-provided source of randomness:

void init(int size)

A third approach initializes a parameter generator object using algorithm-specific semantics, which are represented by a set of algorithm-specific parameter generation values supplied in an AlgorithmParameterSpec object:

void init(AlgorithmParameterSpec genParamSpec,
          SecureRandom random)
void init(AlgorithmParameterSpec genParamSpec)

To generate Diffie-Hellman system parameters, for example, the parameter generation values usually consist of the size of the prime modulus and the size of the random exponent, both specified in number of bits. (The Diffie-Hellman algorithm has been part of the JCE since JCE 1.2.)

Generating Algorithm Parameters

Once you have created and initialized an AlgorithmParameterGenerator object, you can use the generateParameters method to generate the algorithm parameters:

AlgorithmParameters generateParameters()

Key Interfaces

The Key interface is the top-level interface for all opaque keys. It defines the functionality shared by all opaque key objects.

An opaque key representation is one in which you have no direct access to the key material that constitutes a key. In other words: "opaque" gives you limited access to the key--just the three methods defined by the Key interface (see below): getAlgorithm, getFormat, and getEncoded. This is in contrast to a transparent representation, in which you can access each key material value individually, through one of the get methods defined in the corresponding specification class.

All opaque keys have three characteristics:

An Algorithm

The key algorithm for that key. The key algorithm is usually an encryption or asymmetric operation algorithm (such as DSA or RSA), which will work with those algorithms and with related algorithms (such as MD5 with RSA, SHA-1 with RSA, etc.) The name of the algorithm of a key is obtained using this method:

String getAlgorithm()

An Encoded Form

The external encoded form for the key used when a standard representation of the key is needed outside the Java Virtual Machine, as when transmitting the key to some other party. The key is encoded according to a standard format (such as X.509 or PKCS #8), and is returned using the method:

byte[] getEncoded()

A Format

The name of the format of the encoded key. It is returned by the method:

String getFormat()

Keys are generally obtained through key generators, certificates, key specifications (using a KeyFactory), or a KeyStore implementation accessing a keystore database used to manage keys.

It is possible to parse encoded keys, in an algorithm-dependent manner, using a KeyFactory.

It is also possible to parse certificates, using a CertificateFactory.

Here is a list of interfaces which extend the Key interface in the java.security.interfaces package:

- DHPPrivateKey
- DHPublicKey
- DSAPrivateKey
- DSAPublicKey
- PBEKey
- PublicKey
- RSAMultiPrimePrivateCrtKey
- RSAPrivateCrtKey
- RSAPrivateKey
- RSAPublicKey
The PublicKey and PrivateKey Interfaces

The PublicKey and PrivateKey interfaces (which both extend the Key interface) are methodless interfaces, used for type-safety and type-identification.

Key Specification Interfaces and Classes

Key specifications are transparent representations of the key material that constitutes a key. If the key is stored on a hardware device, its specification may contain information that helps identify the key on the device.

A transparent representation of keys means that you can access each key material value individually, through one of the get methods defined in the corresponding specification class. For example, DSAPrivateKeySpec defines getX, getP, getQ, and getG methods, to access the private key x, and the DSA algorithm parameters used to calculate the key: the prime p, the sub-prime q, and the base g.

This representation is contrasted with an opaque representation, as defined by the Key interface, in which you have no direct access to the key material fields. In other words, an "opaque" representation gives you limited access to the key--just the three methods defined by the Key interface: getAlgorithm, getFormat, and getEncoded.

A key may be specified in an algorithm-specific way, or in an algorithm-independent encoding format (such as ASN.1). For example, a DSA private key may be specified by its components x, p, q, and g (see DSAPrivateKeySpec), or it may be specified using its DER encoding (see PKCS8EncodedKeySpec).

In the following sections, we discuss the key specification interfaces and classes in the java.security.spec package.

The KeySpec Interface

This interface contains no methods or constants. Its only purpose is to group and provide type safety for all key specifications. All key specifications must implement this interface.

The DSAPrivateKeySpec Class

This class (which implements the KeySpec interface) specifies a DSA private key with its associated parameters. DSAPrivateKeySpec has the following methods:

- BigInteger getX()
- BigInteger getP()
- BigInteger getQ()
- BigInteger getG()

These methods return the private key x, and the DSA algorithm parameters used to calculate the key: the prime p, the sub-prime q, and the base g.

The DSAPublicKeySpec Class

This class (which implements the KeySpec interface) specifies a DSA public key with its associated parameters. DSAPublicKeySpec has the following methods:

- BigInteger getY()
- BigInteger getP()
- BigInteger getQ()
- BigInteger getG()

These methods return the public key y, and the DSA algorithm parameters used to calculate the key: the prime p, the sub-prime q, and the base g.

The RSAPrivateKeySpec Class
This class (which implements the `KeySpec` interface) specifies an RSA private key. `RSAPrivateKeySpec` has the following methods:

```java
BigInteger getModulus()
BigInteger getPrivateExponent()
```

These methods return the RSA modulus \( n \) and private exponent \( d \) values that constitute the RSA private key.

### The `RSAPrivateCrtKeySpec` Class

This class (which extends the `RSAPrivateKeySpec` class) specifies an RSA private key, as defined in the PKCS #1 standard, using the Chinese Remainder Theorem (CRT) information values. `RSAPrivateCrtKeySpec` has the following methods (in addition to the methods inherited from its superclass `RSAPrivateKeySpec`):

```java
BigInteger getPublicExponent()
BigInteger getPrimeP()
BigInteger getPrimeQ()
BigInteger getPrimeExponentP()
BigInteger getPrimeExponentQ()
BigInteger getCrtCoefficient()
```

These methods return the public exponent \( e \) and the CRT information integers: the prime factor \( p \) of the modulus \( n \), the prime factor \( q \) of \( n \), the exponent \( d \) mod \((p-1)\), the exponent \( d \) mod \((q-1)\), and the Chinese Remainder Theorem coefficient \((inverse of q) \mod p\).

An RSA private key logically consists of only the modulus and the private exponent. The presence of the CRT values is intended for efficiency.

### The `RSAMultiPrimeCrtKeySpec` Class

This class (which extends the `RSAPrivateKeySpec` class) specifies an RSA multi-prime private key, as defined in the PKCS#1 v2.1, using the Chinese Remainder Theorem (CRT) information values. `RSAMultiPrimePrivateCrtKeySpec` has the following methods (in addition to the methods inherited from its superclass `RSAPrivateKeySpec`):

```java
BigInteger getPublicExponent()
BigInteger getPrimeP()
BigInteger getPrimeQ()
BigInteger getPrimeExponentP()
BigInteger getPrimeExponentQ()
BigInteger getCrtCoefficient()
RSAOtherPrimeInfo[] getOtherPrimeInfo()
```

These methods return the public exponent \( e \) and the CRT information integers: the prime factor \( p \) of the modulus \( n \), the prime factor \( q \) of \( n \), the exponent \( d \) mod \((p-1)\), the exponent \( d \) mod \((q-1)\), and the Chinese Remainder Theorem coefficient \((inverse of q) \mod p\).

Method `getOtherPrimeInfo` returns a copy of the `otherPrimeInfo` (defined in PKCS#1 v2.1) or null if there are only two prime factors (\( p \) and \( q \)).

An RSA private key logically consists of only the modulus and the private exponent. The presence of the CRT values is intended for efficiency.

### The `RSAPublicKeySpec` Class

This class (which implements the `KeySpec` interface) specifies an RSA public key. `RSAPublicKeySpec` has the following methods:

```java
BigInteger getModulus()
BigInteger getPublicExponent()
```

These methods return the RSA modulus \( n \) and public exponent \( e \) values that constitute the RSA public key.
The **EncodedKeySpec Class**

This abstract class (which implements the [KeySpec](#) interface) represents a public or private key in encoded format. Its `getEncoded` method returns the encoded key:

```java
abstract byte[] getEncoded();
```

and its `getFormat` method returns the name of the encoding format:

```java
abstract String getFormat();
```

See the next sections for the concrete implementations `PKCS8EncodedKeySpec` and `X509EncodedKeySpec`.

The **PKCS8EncodedKeySpec Class**

This class, which is a subclass of `EncodedKeySpec`, represents the DER encoding of a private key, according to the format specified in the PKCS #8 standard. Its `getEncoded` method returns the key bytes, encoded according to the PKCS #8 standard. Its `getFormat` method returns the string "PKCS#8".

The **X509EncodedKeySpec Class**

This class, which is a subclass of `EncodedKeySpec`, represents the DER encoding of a public key, according to the format specified in the X.509 standard. Its `getEncoded` method returns the key bytes, encoded according to the X.509 standard. Its `getFormat` method returns the string "X.509".

The **KeyFactory Class**

The `KeyFactory` class is an [engine class](#) designed to provide conversions between opaque cryptographic keys (of type `Key`) and [key specifications](#) (transparent representations of the underlying key material).

Key factories are bi-directional. They allow you to build an opaque key object from a given key specification (key material), or to retrieve the underlying key material of a key object in a suitable format.

Multiple compatible key specifications can exist for the same key. For example, a DSA public key may be specified by its components \( y, p, q, \) and \( g \) (see `DSAPublicKeySpec`), or it may be specified using its DER encoding according to the X.509 standard (see `X509EncodedKeySpec`).

A key factory can be used to translate between compatible key specifications. Key parsing can be achieved through translation between compatible key specifications, e.g., when you translate from `X509EncodedKeySpec` to `DSAPublicKeySpec`, you basically parse the encoded key into its components. For an example, see the end of the [Generating/Verifying Signatures Using Key Specifications and KeyFactory](#) section.

Creating a **KeyFactory Object**

As with all engine classes, the way to get a `KeyFactory` object for a particular type of key algorithm is to call the `getInstance` static factory method on the `KeyFactory` class:

```java
static KeyFactory getInstance(String algorithm)
```

**Note:** The algorithm name is not case-sensitive.

A caller may optionally specify the name of a provider or the `Provider` class, which will guarantee that the implementation of the key factory requested is from the named provider.

```java
static KeyFactory getInstance(String algorithm, String provider)
static KeyFactory getInstance(String algorithm, Provider provider)
```
Converting Between a Key Specification and a Key Object

If you have a key specification for a public key, you can obtain an opaque `PublicKey` object from the specification by using the `generatePublic` method:

```java
PublicKey generatePublic(KeySpec keySpec)
```

Similarly, if you have a key specification for a private key, you can obtain an opaque `PrivateKey` object from the specification by using the `generatePrivate` method:

```java
PrivateKey generatePrivate(KeySpec keySpec)
```

Converting Between a Key Object and a Key Specification

If you have a `Key` object, you can get a corresponding key specification object by calling the `getKeySpec` method:

```java
KeySpec getKeySpec(Key key, Class keySpec)
```

device identifies the specification class in which the key material should be returned. It could, for example, be `DSAPublicKeySpec.class`, to indicate that the key material should be returned in an instance of the `DSAPublicKeySpec` class.

Please see the Examples section for more details.

The CertificateFactory Class

The `CertificateFactory` class is an engine class that defines the functionality of a certificate factory, which is used to generate certificate and certificate revocation list (CRL) objects from their encodings.

A certificate factory for X.509 must return certificates that are an instance of `java.security.cert.X509Certificate`, and CRLs that are an instance of `java.security.cert.X509CRL`.

Creating a CertificateFactory Object

As with all engine classes, the way to get a `CertificateFactory` object for a particular certificate or CRL type is to call the `getInstance` static factory method on the `CertificateFactory` class:

```java
static CertificateFactory getInstance(String type)
```

**Note:** The type name is not case-sensitive.

A caller may optionally specify the name of a provider or the `Provider` class, which will guarantee that the implementation of the certificate factory requested is from the named provider.

```java
static CertificateFactory getInstance(String type, String provider)
static CertificateFactory getInstance(String type, Provider provider)
```

Generating Certificate Objects

To generate a certificate object and initialize it with the data read from an input stream, use the `generateCertificate` method:

```java
final Certificate generateCertificate(InputStream inStream)
```

To return a (possibly empty) collection view of the certificates read from a given input stream, use the `generateCertificates` method:

```java
final Collection generateCertificates(InputStream inStream)
```
Generating CRL Objects

To generate a certificate revocation list (CRL) object and initialize it with the data read from an input stream, use the `generateCRL` method:

```java
final CRL generateCRL(InputStream inStream)
```

To return a (possibly empty) collection view of the CRLs read from a given input stream, use the `generateCRLs` method:

```java
final Collection generateCRLs(InputStream inStream)
```

Generating CertPath Objects

To generate a `CertPath` object and initialize it with data read from an input stream, use one of the following `generateCertPath` methods (with or without specifying the encoding to be used for the data):

```java
final CertPath generateCertPath(InputStream inStream)

final CertPath generateCertPath(InputStream inStream, String encoding)
```

To generate a `CertPath` object and initialize it with a list of certificates, use the following method:

```java
final CertPath generateCertPath(List certificates)
```

To retrieve a list of the `CertPath` encodings supported by this certificate factory, you can call the `getCertPathEncodings` method:

```java
final Iterator getCertPathEncodings()
```

The default encoding will be listed first.

The KeyPair Class

The `KeyPair` class is a simple holder for a key pair (a public key and a private key). It has two public methods, one for returning the private key, and the other for returning the public key:

```java
PrivateKey getPrivate()
PublicKey getPublic()
```

The KeyPairGenerator Class

The `KeyPairGenerator` class is an engine class used to generate pairs of public and private keys.

There are two ways to generate a key pair: in an algorithm-independent manner, and in an algorithm-specific manner. The only difference between the two is the initialization of the object.

Please see the Examples section for examples of calls to the methods documented below.

Creating a KeyPairGenerator

All key pair generation starts with a `KeyPairGenerator`. This generation is done using one of the factory methods on `KeyPairGenerator`:

```java
static KeyPairGenerator getInstance(String algorithm)
static KeyPairGenerator getInstance(String algorithm, String provider)
static KeyPairGenerator getInstance(String algorithm, Provider provider)
```

**Note:** The algorithm name is not case-sensitive.
**Initializing a KeyPairGenerator**

A key pair generator for a particular algorithm creates a public/private key pair that can be used with this algorithm. It also associates algorithm-specific parameters with each of the generated keys.

A key pair generator needs to be initialized before it can generate keys. In most cases, algorithm-independent initialization is sufficient. But in other cases, algorithm-specific initialization is used.

**Algorithm-Independent Initialization**

All key pair generators share the concepts of a keysize and a source of randomness. The keysize is interpreted differently for different algorithms. For example, in the case of the DSA algorithm, the keysize corresponds to the length of the modulus. (See Appendix B: Algorithms for information about the keysizes for specific algorithms.)

An `initialize` method takes two universally shared types of arguments:

```java
void initialize(int keysize, SecureRandom random)
```

Another `initialize` method takes only a `keysize` argument; it uses a system-provided source of randomness:

```java
void initialize(int keysize)
```

Since no other parameters are specified when you call the above algorithm-independent `initialize` methods, it is up to the provider what to do about the algorithm-specific parameters (if any) to be associated with each of the keys.

If the algorithm is a "DSA" algorithm, and the modulus size (keysize) is 512, 768, or 1024, then the "SUN" provider uses a set of precomputed values for the \( p \), \( q \), and \( g \) parameters. If the modulus size is not one of the above values, the "SUN" provider creates a new set of parameters. Other providers might have precomputed parameter sets for more than just the three modulus sizes mentioned above. Still others might not have a list of precomputed parameters at all and instead always create new parameter sets.

**Algorithm-Specific Initialization**

For situations where a set of algorithm-specific parameters already exists (such as "community parameters" in DSA), there are two `initialize` methods that have an `AlgorithmParameterSpec` argument. One also has a `SecureRandom` argument, while the source of randomness is system-provided for the other:

```java
void initialize(AlgorithmParameterSpec params, SecureRandom random)

void initialize(AlgorithmParameterSpec params)
```

See the [Examples](#) section for more details.

**Generating a Key Pair**

The procedure for generating a key pair is always the same, regardless of initialization (and of the algorithm). You always call the following method from `KeyPairGenerator`:

```java
KeyPair generateKeyPair()
```

Multiple calls to `generateKeyPair` will yield different key pairs.

**Key Management**
A database called a "keystore" can be used to manage a repository of keys and certificates. (A certificate is a digitally signed statement from one entity, saying that the public key of some other entity has a particular value.)

**Keystore Location**

The keystore is by default stored in a file named `.keystore` in the user's home directory, as determined by the "user.home" system property. On Solaris systems "user.home" defaults to the user's home directory. On Win32 systems, given user name `uName`, "user.home" defaults to:

- `C:\Winnt\Profiles\uName` on multi-user Windows NT systems
- `C:\Windows\Profiles\uName` on multi-user Windows 95/98/2000 systems
- `C:\Windows` on single-user Windows 95/98/2000 systems

**Keystore Implementation**

The `KeyStore` class supplies well-defined interfaces to access and modify the information in a keystore. It is possible for there to be multiple different concrete implementations, where each implementation is that for a particular type of keystore.

Currently, there are two command-line tools that make use of `KeyStore`: `keytool` and `jarsigner`, and also a GUI-based tool named `policytool`. It is also used by the Policy reference implementation when it processes policy files specifying the permissions (allowed accesses to system resources) to be granted to code from various sources. Since `KeyStore` is publicly available, SDK users can write additional security applications that use it.

There is a built-in default implementation, provided by Sun Microsystems. It implements the keystore as a file, utilizing a proprietary keystore type (format) named "JKS". It protects each private key with its individual password, and also protects the integrity of the entire keystore with a (possibly different) password.

Keystore implementations are provider-based. More specifically, the application interfaces supplied by `KeyStore` are implemented in terms of a "Service Provider Interface" (SPI). That is, there is a corresponding abstract `KeystoreSpi` class, also in the `java.security` package, which defines the SPI methods that "providers" must implement. (The term "provider" refers to a package or a set of packages that supply a concrete implementation of a subset of services that can be accessed by the Java 2 SDK Security API.) Therefore, to provide a keystore implementation clients must implement a "provider" and supply a `KeystoreSpi` subclass implementation, as described in How to Implement a Provider for the Java Cryptography Architecture.

Applications can choose different types of keystore implementations from different providers, using the `getInstance` factory method in the `KeyStore` class. A keystore type defines the storage and data format of the keystore information, and the algorithms used to protect private keys in the keystore and the integrity of the keystore itself. Keystore implementations of different types are not compatible.

The default keystore type is "jks" (the proprietary type of the keystore implementation provided by the "SUN" provider). This is specified by the following line in the security properties file:

```
keystore.type=jks
```

To have tools and other applications use a keystore implementation other than the default keystore, you can change that line to specify a different keystore type. Another solution would be to let users of your tools and applications specify a keystore type, and pass that value to the `getInstance` method of `KeyStore`.

An example of the former approach is the following: If you have a provider package that supplies a keystore implementation for a keystore type called `pkcs12`, change the line to

```
keystore.type=pkcs12
```

**Note:** Keystore type designations are not case-sensitive. For example, "JKS" would be considered the same as "jks".
The KeyStore Class

The KeyStore class is an engine class that supplies well-defined interfaces to access and modify the information in a keystore.

This class represents an in-memory collection of keys and certificates. KeyStore manages two types of entries:

**Key Entry**

This type of keystore entry holds very sensitive cryptographic key information, which is stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain authenticating the corresponding public key.

Private keys and certificate chains are used by a given entity for self-authentication using digital signatures. For example, software distribution organizations digitally sign JAR files as part of releasing and/or licensing software.

**Trusted Certificate Entry**

This type of entry contains a single public key certificate belonging to another party. It is called a trusted certificate because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the subject (owner) of the certificate.

This type of entry can be used to authenticate other parties.

Each entry in a keystore is identified by an "alias" string. In the case of private keys and their associated certificate chains, these strings distinguish among the different ways in which the entity may authenticate itself. For example, the entity may authenticate itself using different certificate authorities, or using different public key algorithms.

Whether keystores are persistent, and the mechanisms used by the keystore if it is persistent, are not specified here. This convention allows use of a variety of techniques for protecting sensitive (e.g., private or secret) keys. Smart cards or other integrated cryptographic engines (SafeKeyper) are one option, and simpler mechanisms such as files may also be used (in a variety of formats).

The main KeyStore methods are described below.

**Creating a KeyStore Object**

As with all engine classes, the way to get a KeyStore object is to call the getInstance static factory method on the KeyStore class:

```java
static KeyStore getInstance(String type)
```

A caller may optionally specify the name of a provider or the Provider class, which will guarantee that the implementation of the type requested is from the named provider:

```java
static KeyStore getInstance(String type, String provider)
static KeyStore getInstance(String type, Provider provider)
```

**Loading a Particular Keystore into Memory**

Before a KeyStore object can be used, the actual keystore data must be loaded into memory via the load method:

```java
final void load(InputStream stream, char[] password)
```

The optional password is used to check the integrity of the keystore data. If no password is supplied, no integrity check is performed.
To create an empty keystore, you pass `null` as the `InputStream` argument to the `load` method.

### Getting a List of the Keystore Aliases

All keystore entries are accessed via unique `aliases`. The `aliases` method returns an enumeration of the alias names in the keystore:

```java
final Enumeration aliases()
```

### Determining Keystore Entry Types

As stated in [The KeyStore Class](#), there are two different types of entries in a keystore.

The following methods determine whether the entry specified by the given alias is a key/certificate or a trusted certificate entry, respectively:

```java
final boolean isKeyEntry(String alias)
final boolean isCertificateEntry(String alias)
```

### Adding/Setting/Deleting Keystore Entries

The `setCertificateEntry` method assigns a certificate to a specified alias:

```java
final void setCertificateEntry(String alias, Certificate cert)
```

If `alias` doesn't exist, a trusted certificate entry with that alias is created. If `alias` exists and identifies a trusted certificate entry, the certificate associated with it is replaced by `cert`.

The `setKeyEntry` methods add (if `alias` doesn't yet exist) or set key entries:

```java
final void setKeyEntry(String alias, Key key, char[] password, Certificate[] chain)
final void setKeyEntry(String alias, byte[] key, Certificate[] chain)
```

In the method with `key` as a byte array, it is the bytes for a key in protected format. For example, in the keystore implementation supplied by the "SUN" provider, the `key` byte array is expected to contain a protected private key, encoded as an `EncryptedPrivateKeyInfo` as defined in the PKCS #8 standard. In the other method, the `password` is the password used to protect the key.

The `deleteEntry` method deletes an entry:

```java
final void deleteEntry(String alias)
```

### Getting Information from the Keystore

The `getKey` method returns the key associated with the given alias. The key is recovered using the given password:

```java
final Key getKey(String alias, char[] password)
```

The following methods return the certificate, or certificate chain, respectively, associated with the given alias:

```java
final Certificate getCertificate(String alias)
final Certificate[] getCertificateChain(String alias)
```

You can determine the name (`alias`) of the first entry whose certificate matches a given
final String getCertificateAlias(Certificate cert)

**Saving the KeyStore**

The in-memory keystore can be saved via the `store` method:

```java
final void store(OutputStream stream, char[] password)
```

The password is used to calculate an integrity checksum of the keystore data, which is appended to the keystore data.

**The SecureRandom Class**

The `SecureRandom` class is an engine class that provides the functionality of a random number generator.

**Creating a SecureRandom Object**

As with all engine classes, the way to get a `SecureRandom` object is to call the `getInstance` static factory method on the `SecureRandom` class:

```java
static SecureRandom getInstance(String algorithm)
```

A caller may optionally specify the name of a provider or the `Provider` class, which will guarantee that the implementation of the random number generation (RNG) algorithm requested is from the named provider:

```java
static final SecureRandom getInstance(String algorithm, String provider)
static final SecureRandom getInstance(String algorithm, Provider provider)
```

**Seeding or Re-Seeding the SecureRandom Object**

The `SecureRandom` implementation attempts to completely randomize the internal state of the generator itself unless the caller follows the call to a `getInstance` method with a call to one of the `setSeed` methods:

```java
synchronized public void setSeed(byte[] seed)
public void setSeed(long seed)
```

Once the `SecureRandom` object has been seeded, it will produce bits as random as the original seeds.

At any time a `SecureRandom` object may be re-seeded using one of the `setSeed` methods. The given seed supplements, rather than replaces, the existing seed; therefore, repeated calls are guaranteed never to reduce randomness.

**Using a SecureRandom Object**

To get random bytes, a caller simply passes an array of any length, which is then filled with random bytes:

```java
synchronized public void nextBytes(byte[] bytes)
```

**Generating Seed Bytes**

If desired, it is possible to invoke the `generateSeed` method to generate a given number of seed bytes (to seed other random number generators, for example):

```java
byte[] generateSeed(int numBytes)
```
The Cipher Class

The Cipher class provides the functionality of a cryptographic cipher used for encryption and decryption. It forms the core of the JCE framework.

Creating a Cipher Object

Like other engine classes in the API, Cipher objects are created using the getInstance factory methods of the Cipher class. A factory method is a static method that returns an instance of a class, in this case, an instance of Cipher, which implements a requested transformation.

To create a Cipher object, you must specify the transformation name. You may also specify which provider you want to supply the implementation of the requested transformation:

    public static Cipher getInstance(String transformation);
    public static Cipher getInstance(String transformation, String provider);

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

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

A transformation is a string that describes the operation (or set of operations) to be performed on the given input, to produce some output. A transformation always includes the name of a cryptographic algorithm (e.g., DES), and may be followed by a mode and padding scheme.

A transformation is of the form:

    o "algorithm/mode/padding" or
    o "algorithm"

For example, the following are valid transformations:

    "DES/CBC/PKCS5Padding"
    "DES"

If no mode or padding is specified, provider-specific default values for the mode and padding scheme are used. For example, the SunJCE provider uses ECB as the default mode, and PKCS5Padding as the default padding scheme for DES, DES-EDE and Blowfish ciphers. This means that in the case of the SunJCE provider,

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

and

    Cipher c1 = Cipher.getInstance("DES");

are equivalent statements.

When requesting a block cipher in stream cipher mode (e.g., DES in CFB or OFB mode), you may optionally specify the number of bits to be processed at a time, by appending this number to the mode name as shown in the "DES/CFB8/NoPadding" and "DES/OFB32/PKCS5Padding" transformations. If no such number is specified, a provider-specific default is used. (For example, the SunJCE provider uses a default of 64 bits.)

Appendix A of this document contains a list of standard names that can be used to specify the
algorithm name, mode, and padding scheme components of a transformation.

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

**Initializing a Cipher Object**

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

- **ENCRYPT_MODE**
  Encryption of data.
- **DECRYPT_MODE**
  Decryption of data.
- **WRAP_MODE**
  Wrapping a Key into bytes so that the key can be securely transported.
- **UNWRAP_MODE**
  Unwrapping of a previously wrapped key into a `java.security.Key` object.

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

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

```java
public void init(int opmode, Key key);
public void init(int opmode, Certificate certificate)
public void init(int opmode, Key key, SecureRandom random);
public void init(int opmode, Certificate certificate, SecureRandom random)
public void init(int opmode, Key key, AlgorithmParameterSpec params);
public void init(int opmode, Key key, AlgorithmParameterSpec params, SecureRandom random);
public void init(int opmode, Key key, AlgorithmParameters params)
public void init(int opmode, Key key, AlgorithmParameters params, SecureRandom random)
```

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

However, if a Cipher object that requires parameters is initialized for decryption, and no parameters are supplied to the `init` method, an `InvalidKeyException` or `InvalidAlgorithmParameterException` exception will be raised, depending on the `init` method that has been used.
See the section about Managing Algorithm Parameters for more details.

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

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

## Encrypting and Decrypting Data

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

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

```java
public byte[] doFinal(byte[] input);
public byte[] doFinal(byte[] input, int inputOffset, int inputLen);
public int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output);
public int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)
```

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

```java
public byte[] update(byte[] input);
public byte[] update(byte[] input, int inputOffset, int inputLen);
public int update(byte[] input, int inputOffset, int inputLen, byte[] output);
public int update(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)
```

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

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

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

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

## Wrapping and Unwrapping Keys

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

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

To `wrap` a Key, first initialize the Cipher object for WRAP_MODE, and then call the following:

```java
public final byte[] wrap(Key key);
```

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

1. the name of the key algorithm, and

2. the type of the wrapped key (one of Cipher.SECRET_KEY, Cipher.PRIVATE_KEY, or Cipher.PUBLIC_KEY).

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

```
public String getAlgorithm();
```

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

```
public final Key unwrap(byte[] wrappedKey,
                      String wrappedKeyAlgorithm,
                      int wrappedKeyType));
```

Here, wrappedKey is the bytes returned from the previous call to wrap, wrappedKeyAlgorithm is the algorithm associated with the wrapped key, and wrappedKeyType is the type of the wrapped key. This must be one of Cipher.SECRET_KEY, Cipher.PRIVATE_KEY, or Cipher.PUBLIC_KEY.

### Managing Algorithm Parameters

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

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

```java
import javax.crypto.*;
import java.security.AlgorithmParameters;

// get cipher object for password-based encryption
Cipher c = Cipher.getInstance("PBEWithMD5AndDES");

// initialize cipher for encryption, without supplying // any parameters. Here, "myKey" is assumed to refer // to an already-generated key. c.init(Cipher.ENCRYPT_MODE, myKey);

// encrypt some data and store away ciphertext // for later decryption byte[] cipherText = c.doFinal("This is just an example".getBytes());

// retrieve parameters generated by underlying cipher // implementation AlgorithmParameters algParams = c.getParameters();

// get parameter encoding and store it away byte[] encodedAlgParams = algParams.getEncoded();
```

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

```java
import javax.crypto.*;
import java.security.AlgorithmParameters;
```
// get parameter object for password-based encryption
AlgorithmParameters algParams;
algParams =
    AlgorithmParameters.getInstance("PBEWithMD5AndDES");

// initialize with parameter encoding from above
algParams.init(encodedAlgParams);

// get cipher object for password-based encryption
Cipher c = Cipher.getInstance("PBEWithMD5AndDES");

// initialize cipher for decryption, using one of the
// init() methods that takes an AlgorithmParameters
// object, and pass it the algParams object from above
  c.init(Cipher.DECRYPT_MODE, myKey, algParams);

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

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

- DES, DES-EDE, and Blowfish, when used in feedback (i.e., CBC, CFB, OFB, or PCBC) mode, use an initialization vector (IV). The javax.crypto.spec.IvParameterSpec class can be used to initialize a Cipher object with a given IV.

- PBEWithMD5AndDES uses a set of parameters, comprising a salt and an iteration count. The javax.crypto.spec.PBEParameterSpec class can be used to initialize a Cipher object implementing PBEWithMD5AndDES with a given salt and iteration count.

Note that you do not have to worry about storing or transferring any algorithm parameters for use by the decryption operation if you use the SealedObject class. This class attaches the parameters used for sealing (encryption) to the encrypted object contents, and uses the same parameters for unsealing (decryption).

Cipher Output Considerations

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

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

    public int getOutputSize(int inputLen)

- The Cipher Stream Classes

JCE introduces the concept of secure streams, which combine an InputStream or OutputStream with a Cipher object. Secure streams are provided by the CipherInputStream and CipherOutputStream classes.

- The CipherInputStream Class

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

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

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

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

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

```java
FileInputStream fis;
FileOutputStream fos;
CipherInputStream cis;

fis = new FileInputStream("/tmp/a.txt");
cis = new CipherInputStream(fis, cipher1);
fos = new FileOutputStream("/tmp/b.txt");
byte[] b = new byte[8];
int i = cis.read(b);
while (i != -1) {
fos.write(b, 0, i);
i = cis.read(b);
}
```

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

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

```java
FileInputStream fis;
FileOutputStream fos;
CipherInputStream cis1, cis2;

fis = new FileInputStream("/tmp/a.txt");
cis1 = new CipherInputStream(fis, cipher1);
cis2 = new CipherInputStream(cis1, cipher2);
fos = new FileOutputStream("/tmp/b.txt");
byte[] b = new byte[8];
int i = cis2.read(b);
while (i != -1) {
fos.write(b, 0, i);
i = cis2.read(b);
}
```

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

**The CipherOutputStream Class**

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

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

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

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

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

```java
FileInputStream fis;
FileOutputStream fos;
CipherOutputStream cos;
fis = new FileInputStream("/tmp/a.txt");
fos = new FileOutputStream("/tmp/b.txt");
cos = new CipherOutputStream(fos, cipher1);
byte[] b = new byte[8];
int i = fis.read(b);
while (i != -1) {
    cos.write(b, 0, i);
    i = fis.read(b);
}
cos.flush();
```

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

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

```java
FileInputStream fis;
FileOutputStream fos;
CipherOutputStream cos1, cos2;
fis = new FileInputStream("/tmp/a.txt");
fos = new FileOutputStream("/tmp/b.txt");
cos1 = new CipherOutputStream(fos, cipher1);
cos2 = new CipherOutputStream(cos1, cipher2);
byte[] b = new byte[8];
int i = fis.read(b);
while (i != -1) {
    cos2.write(b, 0, i);
    i = fis.read(b);
}
cos2.flush();
```
The above program copies the content from file `/tmp/a.txt` to `/tmp/b.txt`, except that the content is first encrypted and then decrypted back before it is written to `/tmp/b.txt`.

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

- `flush` flushes the underlying OutputStream by forcing any buffered output bytes that have already been processed by the encapsulated Cipher object to be written out. Any bytes buffered by the encapsulated Cipher object and waiting to be processed by it will **not** be written out.

- `close` closes the underlying OutputStream and releases any system resources associated with it. It invokes the `doFinal` method of the encapsulated Cipher object, causing any bytes buffered by it to be processed and written out to the underlying stream by calling its `flush` method.

### The KeyGenerator Class

A key generator is used to generate secret keys for symmetric algorithms.

#### Creating a Key Generator

Like other engine classes in the API, KeyGenerator objects are created using the `getInstance` factory methods of the KeyGenerator class. A factory method is a static method that returns an instance of `KeyGenerator` which provides an implementation of the requested key generator.

`getInstance` takes as its argument the name of a symmetric algorithm for which a secret key is to be generated. Optionally, a package provider name may be specified:

```java
public static KeyGenerator getInstance(String algorithm);
public static KeyGenerator getInstance(String algorithm, String provider);
```

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

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

#### Initializing a KeyGenerator Object

A key generator for a particular symmetric-key algorithm creates a symmetric key that can be used with that algorithm. It also associates algorithm-specific parameters (if any) with the generated key.

There are two ways to generate a key: in an algorithm-independent manner, and in an algorithm-specific manner. The only difference between the two is the initialization of the object:

- **Algorithm-Independent Initialization**

  All key generators share the concepts of a `keysize` and a `source of randomness`. There is an `init` method that takes these two universally shared types of arguments. There is also one that takes just a `keysize` argument, and uses a system-provided source of randomness, and one that takes just a source of randomness:
public void init(SecureRandom random);
public void init(int keysize);
public void init(int keysize, SecureRandom random);

Since no other parameters are specified when you call the above algorithm-independent init methods, it is up to the provider what to do about the algorithm-specific parameters (if any) to be associated with the generated key.

- **Algorithm-Specific Initialization**

For situations where a set of algorithm-specific parameters already exists, there are two init methods that have an AlgorithmParameterSpec argument. One also has a SecureRandom argument, while the source of randomness is system-provided for the other:

public void init(AlgorithmParameterSpec params);
public void init(AlgorithmParameterSpec params, SecureRandom random);

In case the client does not explicitly initialize the KeyGenerator (via a call to an init method), each provider must supply (and document) a default initialization.

**Creating a Key**

The following method generates a secret key:

public SecretKey generateKey();

**The SecretKeyFactory Class**

This class represents a factory for secret keys.

Key factories are used to convert keys (opaque cryptographic keys of type java.security.Key) into key specifications (transparent representations of the underlying key material in a suitable format), and vice versa.

A javax.crypto.SecretKeyFactory object operates only on secret (symmetric) keys, whereas a java.security.KeyFactory object processes the public and private key components of a key pair.

Objects of type java.security.Key, of which java.security.PublicKey, java.security.PrivateKey, and javax.crypto.SecretKey are subclasses, are opaque key objects, because you cannot tell how they are implemented. The underlying implementation is provider-dependent, and may be software or hardware based. Key factories allow providers to supply their own implementations of cryptographic keys.

For example, if you have a key specification for a Diffie-Hellman public key, consisting of the public value $y$, the prime modulus $p$, and the base $g$, and you feed the same specification to Diffie-Hellman key factories from different providers, the resulting PublicKey objects will most likely have different underlying implementations.

A provider should document the key specifications supported by its secret key factory. For example, the SecretKeyFactory for DES keys supplied by the "SunJCE" provider supports DESKeySpec as a transparent representation of DES keys, the SecretKeyFactory for DES-ede keys supports DESedeKeySpec as a transparent representation of DES-ede keys, and the SecretKeyFactory for PBE supports PBEKeySpec as a transparent representation of the underlying password.

The following is an example of how to use a SecretKeyFactory to convert secret key data into a SecretKey object, which can be used for a subsequent Cipher operation:

```java
// Note the following bytes are not realistic secret key data
// bytes but are simply supplied as an illustration of using data
// bytes (key material) you already have to build a DESKeySpec.
```
byte[] desKeyData = { (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04, (byte)0x05, (byte)0x06, (byte)0x07, (byte)0x08 }; DESKeySpec desKeySpec = new DESKeySpec(desKeyData); SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES"); SecretKey secretKey = keyFactory.generateSecret(desKeySpec);

In this case, the underlying implementation of secretKey is based on the provider of keyFactory.

An alternative, provider-independent way of creating a functionally equivalent SecretKey object from the same key material is to use the javax.crypto.spec.SecretKeySpec class, which implements the javax.crypto.SecretKey interface:

byte[] desKeyData = { (byte)0x01, (byte)0x02, ...}; SecretKeySpec secretKey = new SecretKeySpec(desKeyData, "DES");

- The SealedObject Class

This class enables a programmer to create an object and protect its confidentiality with a cryptographic algorithm.

Given any object that implements the java.io.Serializable interface, one can create a SealedObject that encapsulates the original object, in serialized format (i.e., a "deep copy"), and seals (encrypts) its serialized contents, using a cryptographic algorithm such as DES, to protect its confidentiality. The encrypted content can later be decrypted (with the corresponding algorithm using the correct decryption key) and de-serialized, yielding the original object.

A typical usage is illustrated in the following code segment: In order to seal an object, you create a SealedObject from the object to be sealed and a fully initialized Cipher object that will encrypt the serialized object contents. In this example, the String "This is a secret" is sealed using the DES algorithm. Note that any algorithm parameters that may be used in the sealing operation are stored inside of SealedObject:

```java
// create Cipher object
// Note: sKey is assumed to refer to an already-generated // secret DES key.
Cipher c = Cipher.getInstance("DES");
c.init(Cipher.ENCRYPT_MODE, sKey);

// do the sealing
SealedObject so = new SealedObject("This is a secret", c);
```

The original object that was sealed can be recovered in two different ways:

- by using a Cipher object that has been initialized with the exact same algorithm, key, padding scheme, etc., that were used to seal the object:

```java
    c.init(Cipher.DECRYPT_MODE, sKey);
    try {
        String s = (String)so.getObject(c);
    } catch (Exception e) {
        // do something
    }
```

This approach has the advantage that the party who unseals the sealed object does not require knowledge of the decryption key. For example, after one party has initialized the cipher object with the required decryption key, it could hand over the cipher object to another party who then unseals the sealed object.

- by using the appropriate decryption key (since DES is a symmetric encryption algorithm, we use the same key for scaling and unscaling):

```java
    try {
        String s = (String)so.getObject(sKey);
    } catch (Exception e) {
        // do something
    }
```
In this approach, the `getObject` method creates a cipher object for the appropriate decryption algorithm and initializes it with the given decryption key and the algorithm parameters (if any) that were stored in the sealed object. This approach has the advantage that the party who unseals the object does not need to keep track of the parameters (e.g., the IV) that were used to seal the object.

- **The KeyAgreement Class**

  The KeyAgreement class provides the functionality of a key agreement protocol. The keys involved in establishing a shared secret are created by one of the key generators (`KeyPairGenerator` or `KeyGenerator`), a `KeyFactory`, or as a result from an intermediate phase of the key agreement protocol.

  **Creating a KeyAgreement Object**

  Each party involved in the key agreement has to create a KeyAgreement object. Like other engine classes in the API, KeyAgreement objects are created using the `getInstance` factory methods of the KeyAgreement class. A factory method is a static method that returns an instance of a class, in this case, an instance of `KeyAgreement` which provides the requested key agreement algorithm.

  `getInstance` takes as its argument the name of a key agreement algorithm. Optionally, a package provider name may be specified:

  ```java
  public static KeyAgreement getInstance(String algorithm);
  public static KeyAgreement getInstance(String algorithm, String provider);
  ```

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

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

  **Initializing a KeyAgreement Object**

  You initialize a KeyAgreement object with your private information. In the case of Diffie-Hellman, you initialize it with your Diffie-Hellman private key. Additional initialization information may contain a source of randomness and/or a set of algorithm parameters. Note that if the requested key agreement algorithm requires the specification of algorithm parameters, and only a key, but no parameters are provided to initialize the KeyAgreement object, the key must contain the required algorithm parameters. (For example, the Diffie-Hellman algorithm uses a prime modulus $p$ and a base generator $g$ as its parameters.)

  To initialize a KeyAgreement object, call one of its `init` methods:

  ```java
  public void init(Key key);
  public void init(Key key, SecureRandom random);
  public void init(Key key, AlgorithmParameterSpec params);
  public void init(Key key, AlgorithmParameterSpec params, SecureRandom random);
  ```

  **Executing a KeyAgreement Phase**

  Every key agreement protocol consists of a number of phases that need to be executed by each party involved in the key agreement.

  To execute the next phase in the key agreement, call the `doPhase` method:
public Key doPhase(Key key, boolean lastPhase);

The key parameter contains the key to be processed by that phase. In most cases, this is the public key of one of the other parties involved in the key agreement, or an intermediate key that was generated by a previous phase. doPhase may return an intermediate key that you may have to send to the other parties of this key agreement, so they can process it in a subsequent phase.

The lastPhase parameter specifies whether or not the phase to be executed is the last one in the key agreement: A value of FALSE indicates that this is not the last phase of the key agreement (there are more phases to follow), and a value of TRUE indicates that this is the last phase of the key agreement and the key agreement is completed, i.e., generateSecret can be called next.

In the example of Diffie-Hellman between two parties (see Appendix F), you call doPhase once, with lastPhase set to TRUE. In the example of Diffie-Hellman between three parties, you call doPhase twice: the first time with lastPhase set to FALSE, the 2nd time with lastPhase set to TRUE.

Generating the Shared Secret

After each party has executed all the required key agreement phases, it can compute the shared secret by calling one of the generateSecret methods:

public byte[] generateSecret();
public int generateSecret(byte[] sharedSecret, int offset);
public SecretKey generateSecret(String algorithm);

The Mac Class

The Mac class provides the functionality of a Message Authentication Code (MAC). Please refer to the code example in Appendix F.

Creating a Mac Object

Like other engine classes in the API, Mac objects are created using the getInstance factory methods of the Mac class. A factory method is a static method that returns an instance of a class, in this case, an instance of Mac which provides the requested MAC algorithm.

getInstance takes as its argument the name of a MAC algorithm. Optionally, a package provider name may be specified:

public static Mac getInstance(String algorithm);

public static Mac getInstance(String algorithm, String provider);

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

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

Initializing a Mac Object

A Mac object is always initialized with a (secret) key and may optionally be initialized with a set of parameters, depending on the underlying MAC algorithm.

To initialize a Mac object, call one of its init methods:
public void init(Key key);
public void init(Key key, AlgorithmParameterSpec params);

You can initialize your Mac object with any (secret-)key object that implements the javax.crypto.SecretKey interface. This could be an object returned by javax.crypto.KeyGenerator.generateKey(), or one that is the result of a key agreement protocol, as returned by javax.crypto.KeyAgreement.generateSecret(), or an instance of javax.crypto.spec.SecretKeySpec.

With some MAC algorithms, the (secret-)key algorithm associated with the (secret-)key object used to initialize the Mac object does not matter (this is the case with the HMAC-MD5 and HMAC-SHA1 implementations of the SunJCE provider). With others, however, the (secret-)key algorithm does matter, and an InvalidKeyException is thrown if a (secret-)key object with an inappropriate (secret-)key algorithm is used.

Computing a MAC

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

To compute the MAC of some data in a single step, call the following doFinal method:

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

To compute the MAC of some data in multiple steps, call one of the update methods:

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

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

    public byte[] doFinal();
    public void doFinal(byte[] output, int outOffset);

How to Make Applications "Exempt" from Cryptographic Restrictions

[Note 1: This section should be ignored by most application developers. It is only for people whose applications may be exported to those few countries whose governments mandate cryptographic restrictions, if it desired that such applications have fewer cryptographic restrictions than those mandated.]

[Note 2: Throughout this section, the term "application" is meant to encompass both applications and applets.]

The JCE framework within J2SE 5 includes an ability to enforce restrictions regarding the cryptographic algorithms and maximum cryptographic strengths available to applets/applications in different jurisdiction contexts (locations). Any such restrictions are specified in "jurisdiction policy files".

Due to import control restrictions by the governments of a few countries, the jurisdiction policy files shipped with the J2SE 5 development kit from Sun Microsystems specify that "strong" but limited cryptography may be used. An "unlimited strength" version of these files indicating no restrictions on cryptographic strengths is available for those living in eligible countries (which is most countries). But only the "strong" version can be imported into those countries whose governments mandate restrictions. The JCE framework will enforce the restrictions specified...
in the installed jurisdiction policy files.

It is possible that the governments of some or all such countries may allow certain applications to become exempt from some or all cryptographic restrictions. For example, they may consider certain types of applications as "special" and thus exempt. Or they may exempt any application that utilizes an "exemption mechanism," such as key recovery. Applications deemed to be exempt could get access to stronger cryptography than that allowed for non-exempt applications in such countries.

In order for an application to be recognized as "exempt" at runtime, it must meet the following conditions:

- It must have a permission policy file bundled with it in a JAR file. The permission policy file specifies what cryptography-related permissions the application has, and under what conditions (if any).
- The JAR file containing the application and the permission policy file must have been signed using a code-signing certificate issued after the application was accepted as exempt.

Below are sample steps required in order to make an application exempt from some or all cryptographic restrictions. This is a basic outline that includes information about what is required by JCE in order to recognize and treat applications as being exempt. You will need to know the exemption requirements of the particular country or countries in which you would like your application to be able to be run but whose governments require cryptographic restrictions. You will also need to know the requirements of a JCE framework vendor that has a process in place for handling exempt applications. Consult such a vendor for further information. (Note: The SunJCE provider does not supply an implementation of the ExemptionMechanismSpi class.)

- Step 1: Write and Compile Your Application Code
- Step 2: Create a Permission Policy File Granting Appropriate Cryptographic Permissions
- Step 3: Prepare for Testing
  - Step 3a: Apply for Government Approval From the Government Mandating Restrictions.
  - Step 3b: Get a Code-Signing Certificate
  - Step 3c: Bundle the Application and Permission Policy File into a JAR file
  - Step 3d: Sign the JAR file
  - Step 3e: Set Up Your Environment Like That of a User in a Restricted Country
  - Step 3f: (only for apps using exemption mechanisms) Install a Provider Implementing the Exemption Mechanism Specified in the Permission Policy File
- Step 4: Test Your Application
- Step 5: Apply for U.S. Government Export Approval If Required
- Step 6: Deploy Your Application

**Special Code Requirements for Applications that Use Exemption Mechanisms**

When an application has a permission policy file associated with it (in the same JAR file) and that permission policy file specifies an exemption mechanism, then when the Cipher `getInstance` method is called to instantiate a Cipher, the JCE code searches the installed providers for one that implements the specified exemption mechanism. If it finds such a provider, JCE instantiates an ExemptionMechanism API object associated with the provider's implementation, and then associates the ExemptionMechanism object with the Cipher returned by `getInstance`. 
After instantiating a Cipher, and prior to initializing it (via a call to the Cipher init method), your code must call the following Cipher method:

```java
public ExemptionMechanism getExemptionMechanism()
```

This call returns the ExemptionMechanism object associated with the Cipher. You must then initialize the exemption mechanism implementation by calling the following method on the returned ExemptionMechanism:

```java
public final void init(Key key)
```

The argument you supply should be the same as the argument of the same types that you will subsequently supply to a Cipher init method.

Once you have initialized the ExemptionMechanism, you can proceed as usual to initialize and use the Cipher.

### Permission Policy Files

In order for an application to be recognized at runtime as being "exempt" from some or all cryptographic restrictions, it must have a permission policy file bundled with it in a JAR file. The permission policy file specifies what cryptography-related permissions the application has, and under what conditions (if any).

**Note:** The permission policy file bundled with an application must be named cryptoPerms.

The format of a permission entry in a permission policy file that accompanies an exempt application is the same as the format for a jurisdiction policy file downloaded with the JDK, which is:

```ini
permission <crypto permission class name>[<alg_name>][[, <exemption mechanism name>][[, <maxKeySize>][[, <AlgorithmParameterSpec class name>,<parameters for constructing an AlgorithmParameterSpec object>]]]];
```

See Appendix D for more information about the jurisdiction policy file format.

### Permission Policy Files for Exempt Applications

Some applications may be allowed to be completely unrestricted. Thus, the permission policy file that accompanies such an application usually just needs to contain the following:

```java
grant {
   // There are no restrictions to any algorithms.
   permission javax.crypto.CryptoAllPermission;
};
```

If an application just uses a single algorithm (or several specific algorithms), then the permission policy file could simply mention that algorithm (or algorithms) explicitly, rather than granting CryptoAllPermission. For example, if an application just uses the Blowfish algorithm, the permission policy file doesn't have to grant CryptoAllPermission to all algorithms. It could just specify that there is no cryptographic restriction if the Blowfish algorithm is used. In order to do this, the permission policy file would look like the following:

```java
grant {
   permission javax.crypto.CryptoPermission "Blowfish";
};
```

### Permission Policy Files for Applications Exempt Due to Exemption Mechanisms

If an application is considered "exempt" if an exemption mechanism is enforced, then the permission policy file that accompanies the application must specify one or more exemption mechanisms. At runtime, the application will be considered exempt if any of
those exemption mechanisms is enforced. Each exemption mechanism must be specified in a permission entry that looks like the following:

```java
// No algorithm restrictions if specified
// exemption mechanism is enforced.
permission javax.crypto.CryptoPermission *,
"<ExemptionMechanismName>";
```

where `<ExemptionMechanismName>` specifies the name of an exemption mechanism. The list of possible exemption mechanism names includes:

- KeyRecovery
- KeyEscrow
- KeyWeakening

As an example, suppose your application is exempt if either key recovery or key escrow is enforced. Then your permission policy file should contain the following:

```java
grant {
    // No algorithm restrictions if KeyRecovery is enforced.
    permission javax.crypto.CryptoPermission *,
    "KeyRecovery";
    // No algorithm restrictions if KeyEscrow is enforced.
    permission javax.crypto.CryptoPermission *,
    "KeyEscrow";
};
```

Note: Permission entries that specify exemption mechanisms should not also specify maximum key sizes. The allowed key sizes are actually determined from the installed exempt jurisdiction policy files, as described in the next section.

### How Bundled Permission Policy Files Affect Cryptographic Permissions

At runtime, when an application instantiates a Cipher (via a call to its `getInstance` method) and that application has an associated permission policy file, JCE checks to see whether the permission policy file has an entry that applies to the algorithm specified in the `getInstance` call. If it does, and the entry grants CryptoAllPermission or does not specify that an exemption mechanism must be enforced, it means there is no cryptographic restriction for this particular algorithm.

If the permission policy file has an entry that applies to the algorithm specified in the `getInstance` call and the entry does specify that an exemption mechanism must be enforced, then the exempt jurisdiction policy file(s) are examined. If the exempt permissions include an entry for the relevant algorithm and exemption mechanism, and that entry is implied by the permissions in the permission policy file bundled with the application, and if there is an implementation of the specified exemption mechanism available from one of the registered providers, then the maximum key size and algorithm parameter values for the Cipher are determined from the exempt permission entry.

If there is no exempt permission entry implied by the relevant entry in the permission policy file bundled with the application, or if there is no implementation of the specified exemption mechanism available from any of the registered providers, then the application is only allowed the standard default cryptographic permissions.

### Code Examples

#### Computing a MessageDigest Object
First create the message digest object, as in the following example:

```java
MessageDigest sha = MessageDigest.getInstance("SHA-1");
```

This call assigns a properly initialized message digest object to the `sha` variable. The implementation implements the Secure Hash Algorithm (SHA-1), as defined in the National Institute for Standards and Technology's (NIST) FIPS 180-1 document. See Appendix A for a complete discussion of standard names and algorithms.

Next, suppose we have three byte arrays, `i1`, `i2` and `i3`, which form the total input whose message digest we want to compute. This digest (or "hash") could be calculated via the following calls:

```java
sha.update(i1);
sha.update(i2);
sha.update(i3);
byte[] hash = sha.digest();
```

An equivalent alternative series of calls would be:

```java
sha.update(i1);
sha.update(i2);
byte[] hash = sha.digest(i3);
```

After the message digest has been calculated, the message digest object is automatically reset and ready to receive new data and calculate its digest. All former state (i.e., the data supplied to `update` calls) is lost.

Some hash implementations may support intermediate hashes through cloning. Suppose we want to calculate separate hashes for:

- `i1`
- `i1` and `i2`
- `i1`, `i2`, and `i3`

A way to do it is:

```java
/* compute the hash for i1 */
sha.update(i1);
byte[] i1Hash = sha.clone().digest();

/* compute the hash for i1 and i2 */
sha.update(i2);
byte[] i12Hash = sha.clone().digest();

/* compute the hash for i1, i2 and i3 */
sha.update(i3);
byte[] i123hash = sha.digest();
```

This code works only if the SHA-1 implementation is cloneable. While some implementations of message digests are cloneable, others are not. To determine whether or not cloning is possible, attempt to clone the `MessageDigest` object and catch the potential exception as follows:

```java
try {
    // try and clone it
    /* compute the hash for i1 */
    sha.update(i1);
    byte[] i1Hash = sha.clone().digest();
    ...
    byte[] i123hash = sha.digest();
} catch (CloneNotSupportedException cnse) {
    // do something else, such as the code shown below
}
```

If a message digest is not cloneable, the other, less elegant way to compute intermediate digests is to create several digests. In this case, the number of intermediate digests to be computed must be known in advance:

```java
MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
MessageDigest sha12 = MessageDigest.getInstance("SHA-1");
MessageDigest sha123 = MessageDigest.getInstance("SHA-1");
```
byte[] i1Hash = sha1.digest(i1);
sha12.update(i1);
byte[] i12Hash = sha12.digest(i2);
sha123.update(i1);
sha123.update(i2);
byte[] i123Hash = sha123.digest(i3);

Generating a Pair of Keys

In this example we will generate a public-private key pair for the algorithm named "DSA" (Digital Signature Algorithm). We will generate keys with a 1024-bit modulus, using a user-derived seed, called userSeed. We don't care which provider supplies the algorithm implementation.

Creating the Key Pair Generator

The first step is to get a key pair generator object for generating keys for the DSA algorithm:

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

Initializing the Key Pair Generator

The next step is to initialize the key pair generator. In most cases, algorithm-independent initialization is sufficient, but in some cases, algorithm-specific initialization is used.

Algorithm-Independent Initialization

All key pair generators share the concepts of a keysize and a source of randomness. A KeyPairGenerator class initialize method has these two types of arguments. Thus, to generate keys with a keysize of 1024 and a new SecureRandom object seeded by the userSeed value, you can use the following code:

SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
random.setSeed(userSeed);
keyGen.initialize(1024, random);

Since no other parameters are specified when you call the above algorithm-independent initialize method, it is up to the provider what to do about the algorithm-specific parameters (if any) to be associated with each of the keys. The provider may use precomputed parameter values or may generate new values.

Algorithm-Specific Initialization

For situations where a set of algorithm-specific parameters already exists (such as "community parameters" in DSA), there are two initialize methods that have an AlgorithmParameterSpec argument. Suppose your key pair generator is for the "DSA" algorithm, and you have a set of DSA-specific parameters, \( p \), \( q \), and \( g \), that you would like to use to generate your key pair. You could execute the following code to initialize your key pair generator (recall that DSAParameterSpec is an AlgorithmParameterSpec):

```
DSAPrimarySpec dsaSpec = new DSAPrimarySpec(p, q, g);
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
random.setSeed(userSeed);
keyGen.initialize(dsaSpec, random);
```

Note: The parameter named \( p \) is a prime number whose length is the modulus length ("size"). Therefore, you don't need to call any other method to specify the modulus length.
Generating the Pair of Keys

The final step is generating the key pair. No matter which type of initialization was used (algorithm-independent or algorithm-specific), the same code is used to generate the key pair:

```java
KeyPair pair = keyGen.generateKeyPair();
```

Generating and Verifying a Signature Using Generated Keys

The following signature generation and verification examples use the key pair generated in the key pair example above.

Generating a Signature

We first create a `signature` object:

```java
Signature dsa = Signature.getInstance("SHA1withDSA");
```

Next, using the key pair generated in the key pair example, we initialize the object with the private key, then sign a byte array called `data`.

```java
/* Initializing the object with a private key */
PrivateKey priv = pair.getPrivate();
dsa.initSign(priv);

/* Update and sign the data */
dsa.update(data);
byte[] sig = dsa.sign();
```

Verifying a Signature

Verifying the signature is straightforward. (Note that here we also use the key pair generated in the key pair example.)

```java
/* Initializing the object with the public key */
PublicKey pub = pair.getPublic();
dsa.initVerify(pub);

/* Update and verify the data */
dsa.update(data);
boolean verifies = dsa.verify(sig);
System.out.println("signature verifies: " + verifies);
```

Generating/Verifying Signatures Using Key Specifications and KeyFactory

Suppose that, rather than having a public/private key pair (as, for example, was generated in the key pair example above), you simply have the components of your DSA private key: \(x\) (the private key), \(p\) (the prime), \(q\) (the sub-prime), and \(g\) (the base).

Further suppose you want to use your private key to digitally sign some data, which is in a byte array named `someData`. You would do the following steps, which also illustrate creating a key specification and using a key factory to obtain a `PrivateKey` from the key specification (`initSign` requires a `PrivateKey`):

```java
DSAPrivateKeySpec dsaPrivKeySpec = new DSAPrivateKeySpec(x, p, q, g);

KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PrivateKey privKey = keyFactory.generatePrivate(dsaPrivKeySpec);

Signature sig = Signature.getInstance("SHA1withDSA");
sig.initSign(privKey);
sig.update(someData);
byte[] signature = sig.sign();
```

Suppose Alice wants to use the data you signed. In order for her to do so, and to verify your signature,
you need to send her three things:

1. the data,
2. the signature, and
3. the public key corresponding to the private key you used to sign the data.

You can store the `someData` bytes in one file, and the `signature` bytes in another, and send those to Alice.

For the public key, assume, as in the signing example above, you have the components of the DSA public key corresponding to the DSA private key used to sign the data. Then you can create a DSAPublicKeySpec from those components:

```java
dsapublicKeySpec = new DSAPublicKeySpec(y, p, q, g);
```

You still need to extract the key bytes so that you can put them in a file. To do so, you can first call the `generatePublic` method on the DSA key factory already created in the example above:

```java
PublicKey pubKey = keyFactory.generatePublic(dsapublicKeySpec);
```

Then you can extract the (encoded) key bytes via the following:

```java
byte[] encKey = pubKey.getEncoded();
```

You can now store these bytes in a file, and send it to Alice along with the files containing the data and the signature.

Now, assume Alice has received these files, and she copied the data bytes from the data file to a byte array named `data`, the signature bytes from the signature file to a byte array named `signature`, and the encoded public key bytes from the public key file to a byte array named `encodedPubKey`.

Alice can now execute the following code to verify the signature. The code also illustrates how to use a key factory in order to instantiate a DSA public key from its encoding (`initVerify` requires a `PublicKey`).

```java
X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encodedPubKey);
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
Signature sig = Signature.getInstance("SHA1withDSA");
sig.initVerify(pubKey);
sig.update(data);
sig.verify(signature);
```

Note: In the above, Alice needed to generate a `PublicKey` from the encoded key bits, since `initVerify` requires a `PublicKey`. Once she has a `PublicKey`, she could also use the `KeyFactory` `getKeySpec` method to convert it to a DSAPublicKeySpec so that she can access the components, if desired, as in:

```java
dsapublicKeySpec = (DSAPublicKeySpec) keyFactory.getKeySpec(pubKey, DSAPublicKeySpec.class);
```

Now she can access the DSA public key components `y`, `p`, `q`, and `g` through the corresponding "get" methods on the DSAPublicKeySpec class (`getY`, `getP`, `getQ`, and `getG`).

### Determining If Two Keys Are Equal

In many cases you would like to know if two keys are equal; however, the default method `java.lang.Object.equals` may not give the desired result. The most provider-independent approach is to compare the encoded keys. If this comparison isn't appropriate (for example, when comparing an `RSAPrivateKey` and an `RSAPrivateCrtKey`), you should compare each component. The following code demonstrates this idea:

```java
static boolean keysEqual(Key key1, Key key2) {
    if (key1.equals(key2)) {
...
return true;
}
if (Arrays.equals(key1.getEncoded(), key2.getEncoded())) {
    return true;
}

// More code for different types of keys here.
// For example, the following code can check if
// an RSAPrivateKey and an RSAPrivateCrtKey are equal:
// if ((key1 instanceof RSAPrivateKey) &&
//     (key2 instanceof RSAPrivateKey)) {
//     if ((key1.getModulus().equals(key2.getModulus())) &&
//         (key1.getPrivateExponent().equals(
//                                      key2.getPrivateExponent()))) {
//         return true;
//     }
// }

return false;

Reading Base64-Encoded Certificates

The following example reads a file with Base64-encoded certificates, which are each bounded at the beginning by

-----BEGIN CERTIFICATE-----

and at the end by

-----END CERTIFICATE-----

We convert the FileInputStream (which does not support mark and reset) to a
ByteArrayInputStream (which supports those methods), so that each call to generateCertificate consumes only one certificate, and the read position of the input stream is positioned to the next certificate in the file:

FileInputStream fis = new FileInputStream(filename);
BufferedInputStream bis = new BufferedInputStream(fis);
CertificateFactory cf = CertificateFactory.getInstance("X.509");
while (bis.available() > 0) {
    Certificate cert = cf.generateCertificate(bis);
    System.out.println(cert.toString());
}

Parsing a Certificate Reply

The following example parses a PKCS #7-formatted certificate reply stored in a file and extracts all the certificates from it:

FileInputStream fis = new FileInputStream(filename);
CertificateFactory cf = CertificateFactory.getInstance("X.509");
Collection c = cf.generateCertificates(fis);
Iterator i = c.iterator();
while (i.hasNext()) {
    Certificate cert = (Certificate)i.next();
    System.out.println(cert);
}

This section is a short tutorial on how to use some of the major features of the JCE APIs in J2SE 5. Complete sample programs that exercise the APIs can be found in Appendix F of this document.

Using Encryption

This section takes the user through the process of generating a key, creating and initializing a cipher object, encrypting a file, and then decrypting it. Throughout this example, we use the Data Encryption Standard (DES).
Generating a Key

To create a DES key, we have to instantiate a KeyGenerator for DES. We do not specify a provider, because we do not care about a particular DES key generation implementation. Since we do not initialize the KeyGenerator, a system-provided source of randomness will be used to create the DES key:

```java
KeyGenerator keygen = KeyGenerator.getInstance("DES");
SecretKey desKey = keygen.generateKey();
```

After the key has been generated, the same KeyGenerator object can be re-used to create further keys.

Creating a Cipher

The next step is to create a Cipher instance. To do this, we use one of the `getInstance` factory methods of the Cipher class. We must specify the name of the requested transformation, which includes the following components, separated by slashes (/):

- the algorithm name
- the mode (optional)
- the padding scheme (optional)

In this example, we create a DES (Data Encryption Standard) cipher in Electronic Codebook mode, with PKCS #5-style padding. We do not specify a provider, because we do not care about a particular implementation of the requested transformation.

```java
Cipher desCipher;
// Create the cipher
desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
```

We use the generated `desKey` from above to initialize the Cipher object for encryption:

```java
// Initialize the cipher for encryption
desCipher.init(Cipher.ENCRYPT_MODE, desKey);
// Our cleartext
byte[] cleartext = "This is just an example".getBytes();
// Encrypt the cleartext
byte[] ciphertext = desCipher.doFinal(cleartext);
// Initialize the same cipher for decryption
desCipher.init(Cipher.DECRYPT_MODE, desKey);
// Decrypt the ciphertext
byte[] cleartext1 = desCipher.doFinal(ciphertext);
```

cleartext and cleartext1 are identical.

Using Password-Based Encryption

In this example, we prompt the user for a password from which we derive an encryption key.

It would seem logical to collect and store the password in an object of type `java.lang.String`. However, here's the caveat: Objects of type `String` are immutable, i.e., there are no methods defined that allow you to change (overwrite) or zero out the contents of a `String` after usage. This feature makes `String` objects unsuitable for storing security sensitive information such as user passwords. You should always collect and store security sensitive information in a char array instead.
For that reason, the `javax.crypto.spec.PBEKeySpec` class takes (and returns) a password as a char array.

The following method is an example of how to collect a user password as a char array:

```java
/**
 * Reads user password from given input stream.
 */
public char[] readPasswd(InputStream in) throws IOException {
    char[] lineBuffer;
    char[] buf;
    int i;

    buf = lineBuffer = new char[128];
    int room = buf.length;
    int offset = 0;
    int c;

    loop: while (true) {
        switch (c = in.read()) {
            case -1:
            case '\n':
                break loop;
            case '\r':
                int c2 = in.read();
                if ((c2 != '\n') && (c2 != -1)) {
                    if (!(in instanceof PushbackInputStream)) {
                        in = new PushbackInputStream(in);
                    }
                    ((PushbackInputStream)in).unread(c2);
                } else
                    break loop;
            default:
                if (--room < 0) {
                    buf = new char[offset + 128];
                    room = buf.length - offset - 1;
                    System.arraycopy(lineBuffer, 0, buf, 0, offset);
                    Arrays.fill(lineBuffer, ' ');
                    lineBuffer = buf;
                }
                buf[offset++] = (char) c;
                break;
        }
    }

    if (offset == 0) {
        return null;
    }

    char[] ret = new char[offset];
    System.arraycopy(buf, 0, ret, 0, offset);
    Arrays.fill(buf, ' ');

    return ret;
}
```

In order to use Password-Based Encryption (PBE) as defined in PKCS #5, we have to specify a `salt` and an `iteration count`. The same salt and iteration count that are used for encryption must be used for decryption:

```java
PBEKeySpec pbeKeySpec;
PBEParameterSpec pbeParamSpec;
SecretKeyFactory keyFac;

// Salt
byte[] salt = {
    (byte)0xc7, (byte)0x73, (byte)0x21, (byte)0x8c,
    (byte)0x7e, (byte)0xc8, (byte)0xee, (byte)0x99
};

// Iteration count
int count = 20;

// Create PBE parameter set
```
pbeParamSpec = new PBEParameterSpec(salt, count);

// Prompt user for encryption password.
// Collect user password as char array (using the
// "readPasswd" method from above), and convert
// it into a SecretKey object, using a PBE key
// factory.
System.out.print("Enter encryption password: ");
System.out.flush();
pbeKeySpec = new PBEKeySpec(readPasswd(System.in));
keyFac = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
SecretKey pbeKey = keyFac.generateSecret(pbeKeySpec);

// Create PBE Cipher
Cipher pbeCipher = Cipher.getInstance("PBEWithMD5AndDES");

// Initialize PBE Cipher with key and parameters
pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParamSpec);

// Our cleartext
byte[] cleartext = "This is another example".getBytes();

// Encrypt the cleartext
byte[] ciphertext = pbeCipher.doFinal(cleartext);

Using Key Agreement

Please refer to Appendix F for sample programs exercising the Diffie-Hellman key exchange between 2 and 3 parties, respectively.

Appendix A: Standard Names

The Java 2 SDK Security API requires and uses a set of standard names for algorithms, certificate and keystore types. This specification establishes the following names as standard names.

In some cases naming conventions are suggested for forming names that are not explicitly listed, to facilitate name consistency across provider implementations. Such suggestions use items in angle brackets (such as <digest> and <encryption>) as placeholders to be replaced by specific message digest, encryption algorithm, and other names.

Note: Algorithm names are not case-sensitive.

This appendix includes corresponding lists of standard names relevant to the various security subareas:

- Java Certification Path API Programmer's Guide
- JCE Reference Guide
- JSSE Reference Guide

See Appendix B for algorithm specifications.

Message Digest Algorithms

The algorithm names in this section can be specified when generating an instance of MessageDigest.

MD2: The MD2 message digest algorithm as defined in RFC 1319.

MD5: The MD5 message digest algorithm as defined in RFC 1321.

SHA-1: The Secure Hash Algorithm, as defined in Secure Hash Standard, NIST FIPS 180-1.

SHA-256, SHA-384, and SHA-512: New hash algorithms for which the draft Federal Information Processing Standard 180-2, Secure Hash Standard (SHS) is now available. SHA-256 is a 256-bit hash
function intended to provide 128 bits of security against collision attacks, while SHA-512 is a 512-bit hash function intended to provide 256 bits of security. A 384-bit hash may be obtained by truncating the SHA-512 output.

Key and Parameter Algorithms

The algorithm names in this section can be specified when generating an instance of KeyPairGenerator, KeyFactory, AlgorithmParameterGenerator, and AlgorithmParameters.

**DSA:** The Digital Signature Algorithm as defined in FIPS PUB 186.

**RSA:** The RSA encryption algorithm as defined in PKCS #1.

Digital Signature Algorithms

The algorithm names in this section can be specified when generating an instance of Signature.

**ECDSA** (Elliptic Curve Digital Signature Algorithm), an authentication mechanism described in ECC Cipher Suites for TLS (January 2004 draft).

**MD2withRSA:** The MD2 with RSA Encryption signature algorithm which uses the MD2 digest algorithm and RSA to create and verify RSA digital signatures as defined in PKCS #1.

**MD5withRSA:** The MD5 with RSA Encryption signature algorithm which uses the MD5 digest algorithm and RSA to create and verify RSA digital signatures as defined in PKCS #1.

**NONEwithDSA:** This signature algorithm accepts direct raw data to be signed and uses DSA to create and verify DSA digital signatures as defined in FIPS PUB 186. The data must be exactly 20 bytes in length. This algorithms is also known under the alias name of RawDSA.

**SHA1withDSA:** The DSA with SHA-1 signature algorithm which uses the SHA-1 digest algorithm and DSA to create and verify DSA digital signatures as defined in FIPS PUB 186.

**SHA1withRSA:** The signature algorithm with SHA-1 and the RSA encryption algorithm as defined in the OSI Interoperability Workshop, using the padding conventions described in PKCS #1.

**<digest>with<encryption>:** Use this to form a name for a signature algorithm with a particular message digest (such as MD2 or MD5) and algorithm (such as RSA or DSA), just as was done for the explicitly-defined standard names in this section (MD2withRSA, etc.). For the new signature schemes defined in PKCS #1 v 2.0, for which the **<digest>with<encryption>** form is insufficient, **<digest>with<encryption>and<mgf>** can be used to form a name. Here, **<mgf>** should be replaced by a mask generation function such as MGF1. Example: MD5withRSAandMGF1.

Random Number Generation (RNG) Algorithms

The algorithm names in this section can be specified when generating an instance of SecureRandom.

**SHA1PRNG:** The name of the pseudo-random number generation (PRNG) algorithm supplied by the SUN provider. This implementation follows the IEEE P1363 standard, Appendix G.7: "Expansion of source bits", and uses SHA-1 as the foundation of the PRNG. It computes the SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used.

Certificate Types

The types in this section can be specified when generating an instance of CertificateFactory.

**X.509:** The certificate type defined in X.509.

Keystore Types
The types in this section can be specified when generating an instance of `KeyStore`.

**JKS**: The name of the keystore implementation provided by the SUN provider.

**PKCS12**: The transfer syntax for personal identity information as defined in PKCS #12.

### Service Attributes

A cryptographic service is always associated with a particular algorithm or type. For example, a digital signature service is always associated with a particular algorithm (e.g., DSA), and a `CertificateFactory` service is always associated with a particular certificate type (e.g., X.509).

The attributes in this section are for cryptographic services. The service attributes can be used as filters for selecting providers.

Both the attribute name and value are case insensitive.

**KeySize**: The maximum key size that the provider supports for the cryptographic service.

**ImplementedIn**: Whether the implementation for the cryptographic service is done by software or hardware. The value of this attribute is "software" or "hardware".

The JCE API requires and utilizes a set of standard names for algorithms, algorithm modes, and padding schemes. This specification establishes the following names as standard names. It supplements the list of standard names defined in Appendix A in the Java™ Cryptography Architecture API Specification & Reference. Note that algorithm names are treated case-insensitively.

In some cases naming conventions are suggested for forming names that are not explicitly listed, to facilitate name consistency across provider implementations. Such suggestions use items in angle brackets (such as `<digest>` and `<encryption>`) as placeholders to be replaced by specific message digest, encryption algorithm, and other names.

### Cipher

**Algorithm**

The following names can be specified as the `algorithm` component in a `transformation` when requesting an instance of `Cipher`:

- **AES**: Advanced Encryption Standard as specified by NIST in a draft FIPS. Based on the Rijndael algorithm by Joan Daemen and Vincent Rijmen, AES is a 128-bit block cipher supporting keys of 128, 192, and 256 bits.

- **ARCFOUR/RC4**: A stream cipher developed by Ron Rivest. For more information, see K. Kaukonen and R. Thayer, "A Stream Cipher Encryption Algorithm 'Arcfour'", Internet Draft (expired), draft-kaukonen-cipher-arcfour-03.txt.

- **Blowfish**: The block cipher designed by Bruce Schneier.

- **DES**: The Digital Encryption Standard as described in FIPS PUB 46-2.

- **DESede**: Triple DES Encryption (DES-EDE).

- **ECIES (Elliptic Curve Integrated Encryption Scheme)**

- **PBEWith<digest>And<encryption>** or **PBEWith<prf>And<encryption>**: The password-based encryption algorithm (PKCS #5), using the specified message digest (`<digest>`) or pseudo-random function (`<prf>`) and encryption algorithm (`<encryption>`). Examples:
  - **PBEWithMD5AndDES**: The password-based encryption algorithm as defined in: RSA Laboratories, "PKCS #5: Password-Based Encryption..."
Standard," version 1.5, Nov 1993. Note that this algorithm implies CBC as the cipher mode and PKCS5Padding as the padding scheme and cannot be used with any other cipher modes or padding schemes.


- RC2, RC4, and RC5: Variable-key-size encryption algorithms developed by Ron Rivest for RSA Data Security, Inc.

- RSA: The RSA encryption algorithm as defined in PKCS #1.

**Mode**

The following names can be specified as the mode component in a transformation when requesting an instance of Cipher:

- NONE: No mode.
- CBC: Cipher Block Chaining Mode, as defined in FIPS PUB 81.
- CFB: Cipher Feedback Mode, as defined in FIPS PUB 81.
- OFB: Output Feedback Mode, as defined in FIPS PUB 81.
- PCBC: Propagating Cipher Block Chaining, as defined by Kerberos V4.

**Padding**

The following names can be specified as the padding component in a transformation when requesting an instance of Cipher:

- ISO10126Padding. This padding for block ciphers is described in 5.2 Block Encryption Algorithms in the W3C's "XML Encryption Syntax and Processing" document.
- NoPadding: No padding.
- OAEPWith<digest>And<mgf>Padding: Optimal Asymmetric Encryption Padding scheme defined in PKCS #1, where <digest> should be replaced by the message digest and <mgf> by the mask generation function. Example: OAEPWithMD5AndMGF1Padding.
- SSL3Padding: The padding scheme defined in the SSL Protocol Version 3.0, November 18, 1996, section 5.2.3.2 (CBC block cipher):

```c
block-ciphered struct {
    opaque content[SSLCompressed.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

The size of an instance of a GenericBlockCipher must be a multiple of the block cipher's block length.
The padding length, which is always present, contributes to the padding, which implies that if:

\[ \text{sizeof(content)} + \text{sizeof(MAC)} \mod \text{block_length} = 0, \]

padding has to be \((\text{block_length} - 1)\) bytes long, because of the existence of \(\text{padding_length}\).

This make the padding scheme similar (but not quite) to PKCS5Padding, where the padding length is encoded in the padding (and ranges from 1 to block_length). With the SSL scheme, the sizeof(padding) is encoded in the always present \(\text{padding_length}\) and therefore ranges from 0 to block_length-1.

Note that this padding mechanism is not supported by the "SunJCE" provider.

**KeyAgreement**

The following algorithm names can be specified when requesting an instance of **KeyAgreement**:

- **DiffieHellman**: Diffie-Hellman Key Agreement as defined in PKCS #3: Diffie-Hellman Key-Agreement Standard, RSA Laboratories, version 1.4, November 1993.
- **ECMQV** (Elliptic Curve Menezes-Qu-Vanstone) as described in [ECC Cipher Suites For TLS](https://tools.ietf.org/html/rfc4492) (January 2004 draft).

**KeyGenerator**

The following algorithm names can be specified when requesting an instance of **KeyGenerator**:

- AES
- ARCCFOUR/RC4
- Blowfish
- DES
- DESede
- HmacMD5
- HmacSHA1
- HmacSHA256
- HmacSHA384
- HmacSHA512
- RC2

**KeyPairGenerator**

The following algorithm names can be specified when requesting an instance of **KeyPairGenerator**:

- DiffieHellman
SecretKeyFactory

The following algorithm names can be specified when requesting an instance of SecretKeyFactory:

- DES
- DESede
- PBEWith<digest>And<encryption> or PBEWith<prf>And<encryption>: Secret-key factory for use with PKCS #5 password-based encryption, where <digest> is a message digest, <prf> is a pseudo-random function, and <encryption> is an encryption algorithm. Examples: PBEWithMD5AndDES (PKCS #5, v 1.5) and PBEWithHmacSHA1AndDESede (PKCS #5, v 2.0). Note: These both use only the low order 8 bits of each password character.

KeyFactory

The following algorithm names can be specified when requesting an instance of KeyFactory:

- DiffieHellman

AlgorithmParameterGenerator

The following algorithm names can be specified when requesting an instance of AlgorithmParameterGenerator:

- DiffieHellman

AlgorithmParameters

The following algorithm names can be specified when requesting an instance of AlgorithmParameters:

- AES
- Blowfish
- DES
- DESede
- DiffieHellman
- OAEP
- PBE
- PBEWith<digest>And<encryption>
- RC2

MAC

The following algorithm names can be specified when requesting an instance of Mac:

- **HmacSHA1**: The HMAC-SHA1 keyed-hashing algorithm as defined in RFC 2104: "HMAC: Keyed-Hashing for Message Authentication" (February 1997).

- **HmacSHA256**: The HmacSHA256 algorithm as defined in RFC 2104 "HMAC: Keyed-Hashing for Message Authentication" (February 1997) with SHA-256 as the message digest algorithm.

- **HmacSHA384**: The HmacSHA384 algorithm as defined in RFC 2104 "HMAC: Keyed-Hashing for Message Authentication" (February 1997) with SHA-384 as the message digest algorithm.

- **HmacSHA512**: The HmacSHA512 algorithm as defined in RFC 2104 "HMAC: Keyed-Hashing for Message Authentication" (February 1997) with SHA-512 as the message digest algorithm.

- **PBEWith<mac>:** MAC for use with PKCS #5 v 2.0 password-based message authentication standard, where <mac> is a Message Authentication Code algorithm name. Example: PBEWithHmacSHA1.

### Keystore Types

The following types can be specified when requesting an instance of **KeyStore**:

- **JCEKS**: The proprietary keystore type implemented by the "SunJCE" provider.

### Exemption Mechanisms

The following exemption mechanism names can be specified in the permission policy file that accompanies an application considered "exempt" from cryptographic restrictions:

- **KeyEscrow**: An encryption system with a backup decryption capability that allows authorized persons (users, officers of an organization, and government officials), under certain prescribed conditions, to decrypt ciphertext with the help of information supplied by one or more trusted parties who hold special data recovery keys.

- **KeyRecovery**: A method of obtaining the secret key used to lock encrypted data. One use is as a means of providing fail-safe access to a corporation's own encrypted information in times of disaster.

- **KeyWeakening**: A method in which a part of the key can be escrowed or recovered.

### Appendix B: SunJCE Default Keysizes

The SunJCE provider uses the following default keysizes:

- **KeyGenerator**
  - **DES**: 56 bits
  - **Triple DES**: 112 bits
  - **Blowfish**: 56 bits
  - **HmacMD5**: 64 bytes
  - **HmacSHA1**: 64 bytes

- **KeyPairGenerator**
  - **Diffie-Hellman**: 1024 bits
Appendix C: SunJCE Keysize Restrictions

The SunJCE provider enforces the following restrictions on the keysize passed to the initialization methods of the following classes:

- **KeyGenerator**
  
  Restrictions (by algorithm):
  
  - **DES**: keysize must be equal to 56
  - **Triple DES**: keysize must be equal to 112 or 168
    
    Note: A keysize of 112 will generate a Triple DES key with 2 intermediate keys, and a keysize of 168 will generate a Triple DES key with 3 intermediate keys.
    
    - **Blowfish**: keysize must be a multiple of 8, and can only range from 32 to 448, inclusive

- **KeyPairGenerator**
  
  Restrictions (by algorithm):
  
  - **Diffie-Hellman**: keysize must be a multiple of 64, and can only range from 512 to 1024, inclusive

- **AlgorithmParameterGenerator**
  
  Restrictions (by algorithm):
  
  - **Diffie-Hellman**: keysize must be a multiple of 64, and can only range from 512 to 1024, inclusive

Appendix D: Jurisdiction Policy File Format

JCE represents its jurisdiction policy files as J2SE-style policy files with corresponding permission statements. As described in Default Policy Implementation and Policy File Syntax, a J2SE policy file specifies what permissions are allowed for code from specified code sources. A permission represents access to a system resource. In the case of JCE, the "resources" are cryptography algorithms, and code sources do not need to be specified, because the cryptographic restrictions apply to all code.

A jurisdiction policy file consists of a very basic "grant entry" containing one or more "permission entries."

```
grant {
    <permission entries>;
};
```

The format of a permission entry in a jurisdiction policy file is:

```
permission <crypto permission class name>[ <alg_name>
    [[, <exemption mechanism name>]], <maxKeySize>
    [[, <AlgorithmParameterSpec class name>,
        <parameters for constructing an
        AlgorithmParameterSpec object>]]]);
```

A sample jurisdiction policy file that includes restricting the "Blowfish" algorithm to maximum key sizes of 64 bits is:

```
grant {
    permission javax.crypto.CryptoPermission "Blowfish", 64;
```
A permission entry must begin with the word `permission`. The `<crypto permission class name>` in the template above would actually be a specific permission class name, such as `javax.crypto.CryptoPermission`. A crypto permission class reflects the ability of an application/applet to use certain algorithms with certain key sizes in certain environments. There are two crypto permission classes: `CryptoPermission` and `CryptoAllPermission`. The special `CryptoAllPermission` class implies all cryptography-related permissions, that is, it specifies that there are no cryptography-related restrictions.

The `<alg_name>`, when utilized, is a quoted string specifying the standard name (see Appendix A) of a cryptography algorithm, such as "DES" or "RSA".

The `<exemption mechanism name>`, when specified, is a quoted string indicating an exemption mechanism which, if enforced, enables a reduction in cryptographic restrictions. Exemption mechanism names that can be used include "KeyRecovery", "KeyEscrow", and "KeyWeakening".

`<maxKeySize>` is an integer specifying the maximum key size (in bits) allowed for the specified algorithm.

For some algorithms it may not be sufficient to specify the algorithm strength in terms of just a key size. For example, in the case of the "RC5" algorithm, the number of rounds must also be considered. For algorithms whose strength needs to be expressed as more than a key size, the permission entry should also specify an `AlgorithmParameterSpec` class name (such as `javax.crypto.spec.RC5ParameterSpec`) and a list of parameters for constructing the specified `AlgorithmParameterSpec` object.

Items that appear in a permission entry must appear in the specified order. An entry is terminated with a semicolon.

Case is unimportant for the identifiers (`grant`, `permission`) but is significant for the `<crypto permission class name>` or for any string that is passed in as a value.

Note: An "*" can be used as a wildcard for any permission entry option. For example, an "*" (without the quotes) for an `<alg_name>` option means "all algorithms."

---

**Appendix E: Maximum Key Sizes Allowed by "Strong" Jurisdiction Policy Files**

Due to import control restrictions, the jurisdiction policy files shipped with the J2SE 5 Development Kit allow "strong" but limited cryptography to be used. Here are the maximum key sizes allowed by this "strong" version of the jurisdiction policy files:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Maximum Key Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>64</td>
</tr>
<tr>
<td>DESede</td>
<td>*</td>
</tr>
<tr>
<td>RC2</td>
<td>128</td>
</tr>
<tr>
<td>RC4</td>
<td>128</td>
</tr>
<tr>
<td>RC5</td>
<td>128</td>
</tr>
<tr>
<td>RSA</td>
<td>2048</td>
</tr>
<tr>
<td>* (all others)</td>
<td>128</td>
</tr>
</tbody>
</table>

---

**Appendix B: Algorithms**

This appendix specifies details concerning some of the algorithms defined in Appendix A. Any provider supplying an implementation of the listed algorithms must comply with the specifications in this appendix.
To add a new algorithm not specified here, you should first survey other people or companies supplying provider packages to see if they have already added that algorithm, and, if so, use the definitions they published, if available. Otherwise, you should create and make available a template, similar to those found in this Appendix B, with the specifications for the algorithm you provide.

**Specification Template**

The following table shows the fields of the algorithm specifications.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name by which the algorithm is known. This is the name passed to the getInstance method (when requesting the algorithm), and returned by the getAlgorithm method to determine the name of an existing algorithm object. These methods are in the relevant engine classes: Signature, MessageDigest, KeyPairGenerator, and AlgorithmParameterGenerator.</td>
</tr>
<tr>
<td>Type</td>
<td>The type of algorithm: Signature, MessageDigest, KeyPairGenerator, or AlgorithmParameterGenerator.</td>
</tr>
<tr>
<td>Description</td>
<td>General notes about the algorithm, including any standards implemented by the algorithm, applicable patents, etc.</td>
</tr>
<tr>
<td>KeyPair Algorithm (optional)</td>
<td>The keypair algorithm for this algorithm.</td>
</tr>
<tr>
<td>Keysize (optional)</td>
<td>For a keyed algorithm or key generation algorithm: the legal keysizes.</td>
</tr>
<tr>
<td>Size (optional)</td>
<td>For an algorithm parameter generation algorithm: the legal &quot;sizes&quot; for algorithm parameter generation.</td>
</tr>
<tr>
<td>Parameter Defaults (optional)</td>
<td>For a key generation algorithm: the default parameter values.</td>
</tr>
<tr>
<td>Signature Format (optional)</td>
<td>For a Signature algorithm, the format of the signature, that is, the input and output of the verify and sign methods, respectively.</td>
</tr>
</tbody>
</table>

**Algorithm Specifications**

**SHA-1 Message Digest Algorithm**

<table>
<thead>
<tr>
<th>Name</th>
<th>SHA-1</th>
</tr>
</thead>
</table>

**MD2 Message Digest Algorithm**

<table>
<thead>
<tr>
<th>Name</th>
<th>MD2</th>
</tr>
</thead>
</table>

**MD5 Message Digest Algorithm**

<table>
<thead>
<tr>
<th>Name</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>MessageDigest</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Description</td>
<td>The message digest algorithm as defined in RFC 1321. The output of this algorithm is a 128-bit (16 byte) digest.</td>
</tr>
</tbody>
</table>

**The Digital Signature Algorithm**

<table>
<thead>
<tr>
<th>Name</th>
<th>SHA1withDSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Signature</td>
</tr>
<tr>
<td>Description</td>
<td>This algorithm is the signature algorithm described in NIST FIPS 186, using DSA with the SHA-1 message digest algorithm.</td>
</tr>
<tr>
<td>KeyPair Algorithm</td>
<td>DSA</td>
</tr>
<tr>
<td>Signature Format</td>
<td>ASN.1 sequence of two INTEGER values: r and s, in that order: SEQUENCE ::= { r INTEGER, s INTEGER }</td>
</tr>
</tbody>
</table>

**RSA-based Signature Algorithms, with MD2, MD5 or SHA-1**

<table>
<thead>
<tr>
<th>Names</th>
<th>MD2withRSA, MD5withRSA and SHA1withRSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Signature</td>
</tr>
<tr>
<td>Description</td>
<td>These are the signature algorithms that use the MD2, MD5, and SHA-1 message digest algorithms (respectively) with RSA encryption.</td>
</tr>
<tr>
<td>KeyPair Algorithm</td>
<td>RSA</td>
</tr>
<tr>
<td>Signature Format</td>
<td>DER-encoded PKCS #1 block as defined in RSA Laboratory's Public Key Cryptography Standards Note #1. The data encrypted is the digest of the data signed.</td>
</tr>
</tbody>
</table>

**DSA KeyPair Generation Algorithm**

<table>
<thead>
<tr>
<th>Name</th>
<th>DSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>KeyPairGenerator</td>
</tr>
<tr>
<td>Description</td>
<td>This algorithm is the key pair generation algorithm described in NIST FIPS 186 for DSA.</td>
</tr>
<tr>
<td>Keysize</td>
<td>The length, in bits, of the modulus p. This must range from 512 to 1024, and must be a multiple of 64. The default keysize is 1024.</td>
</tr>
<tr>
<td>Parameter Defaults</td>
<td>The following default parameter values are used for keysizes of 512, 768, and 1024 bits:</td>
</tr>
<tr>
<td>512-bit Key Parameters</td>
<td></td>
</tr>
<tr>
<td>SEED = b869c82b 35d70e1b 1ff91b28 e37a62ec dc34409b</td>
<td></td>
</tr>
<tr>
<td>counter = 123</td>
<td></td>
</tr>
<tr>
<td>p = fca682ce 8e12caba 26efccf7 110e526d b078b05e deccdcde b4a208f3 ae1e17ae 01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17</td>
<td></td>
</tr>
<tr>
<td>q = 962edddc 369ca8e bb260ee6 b6a126d9 346e38c5</td>
<td></td>
</tr>
<tr>
<td>g = 678471b2 7a9cf44e e91a49c5 147db1a9 aaf24f0 5a434d64 86931d2d 14271b9e 35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4</td>
<td></td>
</tr>
<tr>
<td>768-bit key parameters</td>
<td></td>
</tr>
<tr>
<td>SEED = 77d0f8c4 dad15eb8 c4f2f8d6 726cef9d 6d5bb399</td>
<td></td>
</tr>
<tr>
<td>counter = 263</td>
<td></td>
</tr>
<tr>
<td>p = e9e64259 9d355f37 c97ffdf35 671208e b078b05e deccdcde b4a208f3 670f8c4 768-bit key parameters</td>
<td></td>
</tr>
<tr>
<td>SEED = 77d0f8c4 dad15eb8 c4f2f8d6 726cef9d 6d5bb399</td>
<td></td>
</tr>
<tr>
<td>counter = 263</td>
<td></td>
</tr>
<tr>
<td>p = e9e64259 9d355f37 c97ffdf35 671208e b078b05e deccdcde b4a208f3 670f8c4</td>
<td></td>
</tr>
</tbody>
</table>
Appendix F: Sample Programs

- Diffie-Hellman Key Exchange between 2 Parties