# Java™ Cryptography Architecture

# API Specification & Reference

***Last Modified: 2 May 1997***

---

# Acknowledgements

The design of the Java Cryptography Architecture benefited from comments made by many contributors from JavaSoft and from the larger Java community. The JDK 1.1 `java.security` APIs were primarily designed and implemented by Benjamin Renaud. He'd like to especially thank those whose feedback had significant impact on the APIs. They are (in alphabetical order): Josh Bloch, Dave Brown, Mary Dageforde, Satish Dharmaraj, Rachel Gollub, Li Gong, James Gosling, Surya Koneru, Jong Lee, Roger Riggs, and Theron Tock. Numerous other people, too many to be listed here, contributed ideas and feedback. Thanks to all of them.

# Introduction

The Java Security API is a new Java core API, 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 Java applications. The first release of Java Security in JDK 1.1 contains a subset of this functionality, including APIs for digital signatures and message digests. In addition, there are abstract interfaces for key management, certificate management and access control. Specific APIs to support X.509 v3 certificates and other certificate formats, and richer functionality in the area of access control, will follow in subsequent JDK releases.

The "Java Cryptography Architecture" (JCA) refers to the framework for accessing and developing cryptographic functionality for the Java Platform. It encompasses the parts of the JDK 1.1 Java Security API related to cryptography (currently, nearly the entire API), as well as a set of conventions and specifications provided in this document. It introduces a "provider" architecture that allows for multiple and interoperable cryptography implementations.

The Java Cryptography Extension (JCE) extends the JCA API to include encryption and key exchange. Together, it and the JCA provide a complete, platform-independent cryptography API. The JCE will be provided in a separate release because it is not currently exportable outside the United States.

This document is both a high-level description and a specification of the JCA API and its default provider, as shipped in the Java Development Kit (JDK) version 1.1. A separate document describing the JCE API will be provided with the JCE release.

Note: The most recent version of this specification can be found on our public Web site at http://java.sun.com/products/jdk/1.1/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: their aim is to let users of the API utilize cryptographic *concepts*, such as digital signatures and message digests, without concern for the implementations or even the algorithms being used to implement these concepts. When complete algorithm-independence is not possible, the JCA provides developers with standardized algorithm-specific APIs. When implementation-independence is not desirable, the JCA lets developers indicate the specific implementations they require.

Implementation independence is achieved using a "provider"-based architecture. The term Cryptography Package Provider ("provider" for short) refers to a package or set of packages that implement specific algorithms, such as the Digital Signature Algorithm (DSA) or the RSA Cryptosystem (RSA). A program may simply request a particular type of object (such as a Signature object) implementing a particular algorithm (such as DSA) 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.

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

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 easily be added.

# Architecture

### Cryptography Package Providers

The Java Cryptography Architecture introduces the notion of a *Cryptography Package Provider* ("provider" for short). This term refers to a package (or a set of packages) that supply a concrete implementation of a subset of the cryptography aspects of the Java Security API. A provider may, for example, contain an implementation of the Digital Signature Algorithm (DSA) and/or the RSA Cryptosystem (RSA).

A program may simply request a particular type of object (such as a Signature object) implementing a particular algorithm (such as DSA) and get an implementation from one of the installed providers. Alternatively, the program can request a specific provider. (Each provider has a name used to refer to it.)

JDK 1.1 comes standard with a default provider, named "SUN". The "SUN" provider package includes:

- An implementation of the Digital Signature Algorithm (NIST FIPS 186).
- An implementation of the MD5 (RFC 1321) and SHA-1 (NIST FIPS 180-1) message digest algorithms.

Each JDK 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.

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 algorithms when no specific provider is requested.

### Key Management

Each Java Virtual Machine has a "system identity scope" (an IdentityScope) which manages a repository of keys, certificates and trust levels. That repository is available to applications that need it for authentication or signing purposes. A default IdentityScope for a persistent database is supplied by the default Cryptography Package Provider, named "SUN". A different IdentityScope could be utilized as the system scope, if desired.

# Concepts

This section covers the major concepts introduced in the API.

### Engine Classes and Algorithms

Engine classes are classes that provide the functionality of a *type* of cryptographic algorithm. The JCA defines a Java class for each engine class. For example, there is a MessageDigest class, a Signature class, and a KeyPairGenerator class. Users of the API request and utilize instances of these engine classes to carry out corresponding operations. A Signature instance is used to sign and verify digital signatures, a MessageDigest instance is used to calculate the message digest of specified data, and a KeyPairGenerator is used to generate pairs of public and private keys suitable for a specified algorithm.

An engine class provides the interface to the functionality of a specific type of algorithm, while its 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 Signature subclass could be that for any kind of signature algorithm, such as SHA-1 with DSA, SHA-1 with RSA, or MD5 with RSA.

As another example, MessageDigest provides access to a message digest algorithm. Its implementations may be that of various message digest algorithms such as SHA-1, MD5, or MD2.

### Implementations and Providers

Implementations for various engine classes are provided by JCA Cryptography Package Providers. Providers are essentially packages that implement one or more engine classes for specific algorithms. For example, the Java Development Kit's default provider, named "SUN", supplies implementations of the DSA signature algorithm and of the MD5 and SHA-1 message digest algorithms. Other providers may define their own implementations of these algorithms or of other algorithms, 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 "SHA/DSA"), and, optionally, the name of the provider whose implementation is desired. The `getInstance` method finds a Signature subclass that satisfies the supplied algorithm and provider parameters. If no provider is specified, `getInstance` searches the registered providers, in preference order, for one with a Signature subclass implementing the specified algorithm. See The Provider Class for more information about registering providers.

# Core Classes and Interfaces

This section provides a discussion of the core classes and interfaces provided in the general release of the Java Cryptography Architecture:

- the Provider and Security classes
- the MessageDigest, Signature, and KeyPairGenerator engine classes
- the Key and related classes

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

- java.security package
- java.security.acl package
- java.security.interfaces package

## The Provider Class

The term "Cryptography Package Provider" ("provider" for short) is used to refer to a package or set of packages that implement specific cryptographic algorithms. 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.

To actually supply implementations of cryptography algorithms, an entity (e.g., a development group) writes the implementation code and creates a subclass of the Provider class. The constructor of the subclass sets the values of various properties that are required for the Java Security API to look up the algorithms or other facilities implemented by the provider. That is, it specifies the names of the classes implementing the algorithms. Note: The Provider subclass can get its information from wherever it wants. Thus, the information can be hard-wired in, or retrieved at runtime, e.g., from a file.

There are several types of algorithms that can be implemented by provider packages: digital signature algorithms (such as DSA or MD5 with RSA), message digest algorithms (such as SHA-1 or MD5) and, with JCE installed, encryption algorithms (such as DES or RSA) and padding schemes (such as PKCS#5).

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.

## 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 whose implementation is desired.

If no provider is specified, `getInstance` searches the registered providers for an implementation of the named algorithm. In any given Java Virtual Machine (JVM), providers are installed in a given *preference order* . That order is the order in which they are searched when no specific provider is requested. For example, suppose there are two providers installed in a JVM, one named "PROVIDER_1" and the other "PROVIDER_2". Further suppose that

- PROVIDER_1 implements SHA/DSA, SHA-1, MD5, DES, and DES3
- PROVIDER_2 implements SHA/DSA, MD5/RSA, MD2/RSA, MD2, MD5, RC4, RC5, DES, and RSA

If PROVIDER_1 has preference order 1 (the highest priority) and PROVIDER_2 has preference order 2, then the following behavior will occur:

- Suppose 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 thus is searched first.
- If we are looking for an MD5/RSA 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.
- Suppose we are looking for an SHA-1/RSA signature algorithm. Since no installed provider implements it, a `NoSuchAlgorithmException` is raised.

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 SHA/DSA implementation from PROVIDER_1 has not received such certification, while the SHA/DSA implementation of PROVIDER_2 has received it.

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

```
Signature dsa = Signature.getInstance("SHA/DSA", "PROVIDER_2");
```

In this case, if "PROVIDER_2" was not installed, a `NoSuchProviderException` would be raised, even if a different 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

Create a `classes` directory in the JDK installation directory, and install the provider classes (the .class files) in that directory. For example, if the JDK is installed in a directory called `jdk1.1.1`, and the classes implementing the provider are in the `COM.acme.provider` package, install the classes in the directory `jdk1.1.1/classes/COM/acme/provider`.

Alternatively, a zip or JAR (Java ARchive) file containing the classes can be located anywhere on your CLASSPATH.

### Configuring the Provider

The next step is to add the provider to your list of approved providers. This is done statically by editing the `java.security` file in the `lib/security` directory of the JDK. Thus, if the JDK is installed in a directory called `jdk1.1.1`, the file would be `jdk1.1.1/lib/security/java.security`. One of the types of properties you can set in `java.security` is of 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.

*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 implemented by the provider.

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, 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.

Note: these methods can only be called from a trusted program. Currently, a "trusted program" is either

- a local application, or
- a trusted applet running in the appletviewer miniature browser that comes with the JDK. An applet is considered "trusted" if
  - it's in a Java ARchive (JAR) file that was signed (using the **javakey** tool) by a trusted (by **javakey**) entity, and
  - the database managed by **javakey** holds a copy of a certificate for the public key of the entity who signed the JAR file (so that the signature can be authenticated).

The appletviewer allows such applets to run with the same full rights as local applications. For information about **javakey**, see the Solaris or Windows documentation.

## Managing Providers

The Security class may be used to query which Providers are installed, as well as to install new ones at runtime.

### Quering Providers

```
public Provider[] getProviders()
```

This method 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.

```
public Provider getProvider(String providerName)
```

This method returns the Provider named `providerName`. It returns `null` if the Provider is not found.

### Adding Providers

```
public void addProvider(Provider provider)
```

This method adds a Provider to the next available preference position. It returns the preference position in which the Provider was added, or -1 if the Provider was not added because it was already installed.

```
public int insertProviderAt(Provider provider, int position)
```

This method adds a new Provider, at a specified position. The position is the preference order in which providers are searched for requested algorithms (if no specific provider is requested). The position is 1-based, that is, 1 is most preferred, followed by 2, and so on. Note that it is not guaranteed that the Provider will be added at the requested position. For example, sometimes it will be legal to add a Provider, but only in the last position, in which case the `position` argument will be ignored. Also, a Provider cannot be added if it is already installed.

This method returns the actual preference position in which the Provider was added, or -1 if the Provider was not added because it was already installed.

#### Removing Providers

```
public void removeProvider(String name)
```

This method removes the Provider with the specified name. It returns silently if the Provider is not installed.

### 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:

```
public static String getProperty(String key)
public 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*. A digest has the following properties:

- It should be computationally infeasible to find another input string that will generate the same digest.
- The digest does 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:

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

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

```
public static MessageDigest getInstance(String algorithm, String 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 making one or more calls to one of the `update` methods:

```
public void update(byte input)
public void update(byte[] input)
public 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:

```
public byte[] digest()
public byte[] digest(byte[] input)
```

A call to the latter method is equivalent to making a call to

```
public 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 (such as Signature).

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 methods, 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:

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

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

```
public static Signature getInstance(String algorithm,
                                    String provider)
```

## Initializing a Signature Object

A Signature object must be initialized before it is used. The initialization method depends on whether the object is first 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:

public 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 the method:

```
public final void initVerify(PublicKey publicKey)
```

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:

```
public final void update(byte b)
public final void update(byte[] data)
public 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 the `sign` method:

```
public final byte[] sign()
```

This returns the signature in a byte array. The signature is encoded as a standard ASN.1/DER sequence of two integers, `r` and `s`. See Appendix B for more information about the use of ASN.1 and DER encoding in the Java Cryptography Architecture.

A call to the `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 from the same signer, if desired, via new calls to `update` and `sign`.

Alternatively, a new call can be made to `initSign` specifying a different private key (to initialize the Signature object for generating a signature from a different entity), 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 whether or not 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. This is done by making one or more calls to one of the `update` methods:

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

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

The signature can then be verified by calling the `verify` method:

```
public final boolean verify(byte[] encodedSignature)
```

The argument must be a byte array containing the signature encoded as a standard ASN.1/DER sequence of two integers, `r` and `s`. This is a standard encoding that is frequently utilized. It is the same as that produced by the `sign` method.

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 the state it was in when previously 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).

# Key Interfaces

The Key interface is the top-level interface for all keys. It defines the functionality shared by all key objects. All keys have three characteristics:

- An Algorithm

  This is 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, Raw DSA, etc.) The name of the algorithm of a key is obtained using the method

  ```
  public String getAlgorithm()
  ```

- An Encoded Form

  This is an 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:

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

- A Format

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

  ```
  public String getFormat()
  ```

Keys are generally obtained through key generators, certificates, or various Identity classes used to manage keys. There are no provisions in this release for the parsing of encoded keys and certificates.

### The PublicKey and PrivateKey Interfaces

The PublicKey and PrivateKey interfaces are method-less interfaces, used for type-safety and type-identification.

## 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:

```
public PrivateKey getPrivate()
public PublicKey getPublic()
```

## The KeyPairGenerator Class

The KeyPairGenerator class is an engine class used to generate pairs of public and private keys. Key generation is an area that sometimes does not lend itself well to algorithm independence. For example, it is possible to generate a DSA key pair specifying key family parameters (p, q and g), while it is not possible to do so for an RSA key pair. That is, those parameters are applicable to DSA but not to RSA.

There are therefore 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 a examples of calls to the methods documented below.

## Creating a KeyPairGenerator

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

```
public static KeyPairGenerator getInstance(String algorithm)
public static KeyPairGenerator getInstance(String algorithm,
    String provider)
```

## Initializing a KeyPairGenerator

A key pair generator needs to be initialized before it can generate keys. In most cases, algorithm-independent initialization is sufficient. But if you want control over parameters specific to a given algorithm, algorithm-specific initialization is utilized.

### Algorithm-Independent Initialization

All key pair generators share the concepts of a "strength" and a source of randomness. The measure of strength is universally shared by all algorithms, though it is interpreted differently for different algorithms. (See Appendix B: Algorithms for information about the strengths for specific algorithms.) The source of randomness must be provided as a SecureRandom object. The KeyPairGenerator class `initialize` method takes these two universally shared types of arguments. Its signature is:

```
public void initialize(int strength, SecureRandom random)
```

Since no other parameters are specified when you call this algorithm-independent `initialize` method, all other values, such as algorithm parameters, public exponent, etc., are defaulted to standard values. Again see Appendix B: Algorithms for more information about defaults for specific algorithms.

### Algorithm-Specific Initialization

It is sometimes desirable to initialize a key pair generator object using algorithm-specific semantics. For example, you may want to initialize a DSA key generator for a given set of parameters `p`, `q` and `g`, or an RSA key generator for a given public exponent `e`.

This is done through algorithm-specific standard interfaces. Rather than calling the algorithm-independent KeyPairGenerator `initialize` method, the key pair generator is cast to an algorithm-specific interface so that one of its specialized parameter initialization methods can be called. An example is the DSAKeyPairGenerator (from `java.security.interfaces`), which provides the following specialized parameter initialization method:

```
public void initialize(DSAParams params, SecureRandom random)
```

See the Examples section for more details.

## Generating a Key Pair

Generating a key pair is always the same, regardless of initialization (and therefore of algorithm). You always call the following method from KeyPairGenerator:

```
public KeyPair generateKeyPair()
```

Multiple calls to generateKeyPair will yield different key pairs.

# Key Management Classes

## The Identity Class

The Identity class is the basic key management entity.

This class represents identities: real-world objects such as people, companies or organizations whose identities can be authenticated using their public keys.

All Identity objects have a name and a public key. Names are immutable. Identities may also be scoped (see IdentityScope). That is, if an Identity is specified to have a particular scope, then the name and public key of the Identity are unique within that scope.

An Identity also has a set of certificates (all certifying its own public key). Note: support for specific certificate formats such as X.509 v3 is not available in JDK 1.1 but will be part of the next JDK release.

The main methods of the Identity class are ones for returning its name, its public key, and its scope:

```
public String getName()

public PublicKey getPublicKey()

public IdentityScope getScope()
```

The name and scope of an identity uniquely identify the Identity. Similarly (since there is a one-to-one mapping between keys and identitities), the key uniquely identifies the Identity.

## The IdentityScope Class

The IdentityScope class is a subclass of the Identity class. It is intended to serve as a general abstraction for repositories of Identity objects (and of objects from subclasses of Identity). Examples include identity databases, identity servers, or PGP key rings. These repositories are used by various mechanisms, such as secure class loaders, to verify classes and assign permissions, or by signing tools to retrieve private keys and generate digital signatures. In general, public key repositories should be kept separate from private key ones.

### Name and Key Scoping

Since an IdentityScope object is itself an Identity, it has a name and can have a scope. It can also optionally have a public key and associated certificates.

There is a one-to-one mapping between keys and identities, and there can only be one copy of one key per scope. That is, no two Identity objects in the same scope can have the same public key or the same name.

An IdentityScope can contain Identity objects of all kinds, including other IdentityScopes. For example, an IdentityScope named "Sun" may contain another scope, "JavaSoft" (along with scopes for other Sun operating companies), which itself may contain another scope, "Security Group". This scope may contain various Identity objects, including one named "Duke".

Clearly there is more than one "Security Group" in the world, and there may be more than one per Virtual Machine, but there is only one in the scope "Sun"/"JavaSoft". Likewise there could be other people named "Duke" in the world, but there is only one which could be scoped "Sun"/"JavaSoft"/"Security Group"/"Duke".

The same scoped uniqueness holds for public keys, since there is a one-to-one correspondence between names and public keys (an Identity object has just one public key). Thus, for example, the public key of the Identity named "Duke" must be unique within its scope. (Similarly, since there is only one private key per public key, this scoped uniqueness holds for private keys as well).

### IdentityScope Methods

All types of Identity objects can be retrieved, added, and removed using the same methods. Note that it is possible, and in fact expected, that different types of identity scopes will apply different policies for their various operations on the various types of Identities. The main IdentityScope methods are:

```
/* Return the identity with the specified name. */
public Identity getIdentity(String name)

/* Add the specified identity to this identity scope. */
public void addIdentity(Identity identity)

/* Remove the specified identity from this identity scope. */
public void removeIdentity(Identity identity)

/* Return an enumeration of all identities in this identity scope. */
public Enumeration identities()
```

### The System Identity Scope

Each Java Virtual Machine has a "system identity scope" (an IdentityScope object) that manages a repository of keys, certificates and trust levels. That repository is available to applications that need it for authentication or signing purposes. A default IdentityScope for a persistent database is supplied by the provider named "SUN". It is `sun.security.provider.IdentityDatabase` (a subclass of IdentityScope). An instance of this class is created every time a Java program is run or an applet viewer is started.

A different IdentityScope could be utilized as the system scope, if desired. An authorized user could change the system scope to be used throughout a JVM by changing the `system.scope` property in the `java.security` file in the `lib/security` directory in the installation directory. Thus, if the JDK is installed in a directory called `jdk1.1.1`, the file would be `jdk1.1.1/lib/security/java.security`. The default entry in the file for the system scope is

```
system.scope=sun.security.provider.IdentityDatabase
```

The value of the `system.scope` property specifies the class to instantiate as the system scope. To change the system scope, you would specify a different IdentityScope than the `sun.security.provider.IdentityDatabase` one. Alternatively, a program could change the system scope just for its own use during its current session (as opposed to more permanently modifying the scope for all users of the JVM). It would do this by calling the `setSystemScope` method of IdentityScope:

```
public void setSystemScope(IdentityScope scope)
```

At any time, you can get the system identity scope by calling the following method:

```
public IdentityScope getSystemScope()
```

In general, public key repositories should be kept separate from private key ones. This is not always easy to do. The system scope will generally be the public database, while the private scope for a given user may be in a private directory such as his or her home directory.

### The Signer Class

The Signer class is a subclass of Identity. It is used for representing an Identity that can sign data. Such an entity must have a "key pair," that is, both a public key and an associated private key. (The private key is required for signing data, and the associated public key is needed for verifying the signature.) The Signer class thus adds a method for setting the key pair:

```
public final void setKeyPair(KeyPair pair)
```

It also adds a method to return the private key (the public key can be returned by the `getPublicKey` method from the Identity class):

```
public PrivateKey getPrivateKey()
```

## The SecureRandom Class

The SecureRandom class is intended to provide a software-based, platform independent, good-quality random number generator.

### Creating a SecureRandom Object

There are two ways to instantiate a SecureRandom instance: either using the default seed mechanism or by providing a seed. Once the SecureRandom object has been seeded, it will produce bits as random as the original seeds. The constructors are:

```
public SecureRandom()

public SecureRandom(byte[] seed)
```

The default seeding mechanism currently implemented in JDK 1.1 is experimental, and applications requiring cryptographically secure random numbers should seed SecureRandom instances using a cryptographically secure seed.

### Using a SecureRandom Object

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

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

### Re-Seeding a SecureRandom Object

At any time a SecureRandom object may be re-seeded using the method:

```
public void setSeed(byte[] seed)
```

This resets the seed. The given seed supplements, rather than replaces, the existing seed. Thus, repeated calls are guaranteed never to reduce randomness.

# Code Examples

### Computing a MessageDigest Object

First create the message digest object, as in the following example:

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

This call assigns a properly initialized message digest object to the `sha` variable. That object will implement 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 calculate via the following calls:

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

An equivalent alternative series of calls would be:

```
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:

```
/* 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 will work only if the SHA-1 implementation is cloneable. While some implementation of message digests are cloneable, others are not. The way to test if an given message digest instance is cloneable is:

```
boolean cloneable = sha instanceof Cloneable;
```

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:

```
MessageDigest i1 = MessageDigest.getMessageDigest("SHA-1");
MessageDigest i12 = MessageDigest.getMessageDigest("SHA-1");
MessageDigest i123 = MessageDigest.getMessageDigest("SHA-1");

byte[] i1Hash = i1.digest(i1);

i12.update(i1);
byte[] i12Hash = i12.digest(i2);

i123.update(i1);
i123.update(i2);
byte[] i123Hash = i123.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 if you want control over parameters specific to the DSA algorithm, algorithm-specific initialization is utilized.

#### Algorithm-Independent Initialization

All key pair generators share the concepts of a "strength" and a source of randomness. The general KeyPairGenerator class `initialize` method has these two types of arguments. Thus, to generate keys with a modulus length of 1024 and a new SecureRandom object seeded by the `userSeed` value, use the following code:

```
keyGen.initialize(1024, new SecureRandom(userSeed));
```

**Algorithm-Specific Initialization**

When you use algorithm-independent initialization, you accept default values the implementation has for any algorithm-specific parameters (other than the strength).

Suppose we don't want to accept the defaults; we have a set of DSA-specific parameters, `p`, `q`, and `g`, that we would like to use to generate our key pair.

Then instead of calling the KeyPairGenerator `initialize` method, we cast the KeyPairGenerator object we obtained via `KeyPairGenerator.getInstance` to an appropriate interface. In our sample case, we asked for a key pair generator for the "DSA" algorithm. Such an algorithm implements the `DSAKeyPairGenerator` interface (from `java.security.interfaces`). So we can cast the generator to a DSAKeyPairGenerator, then call the `initialize` method from that interface to supply DSA-specific parameters.

There is a `java.security.interfaces.DSAParams` interface for grouping DSA-specific parameters. The `initialize` method in DSAKeyPairGenerator takes two arguments: a DSAParams object specifying the `p`, `q`, and `g` parameters, and a SecureRandom object for the source of randomness.

To build the first argument, we construct an instance of a simple class implementing the DSAParams interface and pass it the p, q, and g parameters, represented as BigInteger objects. An example of a simple class named `DSAParamsClass` that implements DSAParams could be defined by the following:

```
import java.math.BigInteger;

class DSAParamsClass implements java.security.interfaces.DSAParams {
  BigInteger p, q, g;

  DSAParamsClass(BigInteger p, BigInteger q, BigInteger g) {
      this.p = p;
      this.q = q;
      this.g = g;
  }

  public BigInteger getP() {
      return p;
  }

  public BigInteger getQ() {
      return q;
  }
```

```
    public BigInteger getG() {
        return g;
    }
}
```

For the second argument to the `initialize` method, we use the same code as was used in the algorithm-independent method to specify the source of randomness.

Assuming the class implementing DSAParams is named `DSAParamsClass` (as above), use the following to initialize the key pair generator:

```
DSAParams dsaParams = new DSAParamsClass(p, q, g);
DSAKeyPairGenerator dsaKeyGen = (DSAKeyPairGenerator)keyGen;
dsaKeyGen.initialize(dsaParams, new SecureRandom(userSeed));
```

(Note: The parameter named p is a prime number whose length is the modulus length. Thus, 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 utilized (algorithm-independent or algorithm-specific), the same code is used to generate the key pair:

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

## Generating a Signature

We first create a signature object:

```
Signature dsa = Signature.getInstance("SHA/DSA");
```

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`.

```
/* 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: here we also use the key pair generated in the key pair example)

```
/* 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);
```

# Appendix A: Standard Names

The Java Security API requires and utilizes a set of standard names for various algorithms, padding schemes, providers, etc. This specification establishes the following names as standard names. See Appendix B for algorithm specifications.

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

MD5: The Message Digest algorithm RSA-MD5, as defined by RSA DSI in RFC 1321.

MD2: The Message Digest algorithm RSA-MD2, as defined by RSA DSI in RFC 1423.

RawDSA: The asymmetric transformation described in NIST FIPS 186, described as the "DSA Sign Operation" and the "DSA Verify Operation", prior to creating a digest. The input to RawDSA is always 20 bytes long.

RSA: The Rivest, Shamir and Adleman AsymmetricCipher algorithm. RSA Encryption as defined in the RSA Laboratory Technical Note PKCS#1.

DSA: Digital Signature Algorithm, as defined in Digital Signature Standard, NIST FIPS 186. This standard defines a digital signature algorithm that uses the RawDSA asymmetric transformation along with the SHA-1 message digest algorithm.

MD5/RSA: The Signature algorithm obtained by combining the RSA AsymmetricCipher algorithm with the MD5 MessageDigest Algorithm.

MD2/RSA: The Signature algorithm obtained by combining the RSA AsymmetricCipher algorithm with the MD2 MessageDigest Algorithm.

SHA-1/RSA: The Signature algorithm obtained by combining the RSA AsymmetricCipher algorithm with the SHA-1 MessageDigest Algorithm.

DES: The Data Encryption Standard, as defined by NIST in FIPS 46-1 and 46-2.

IDEA: The International Data Encryption Algorithm (IDEA) from ASCOM Systec, Switzerland.

RC2: SymmetricCipher algorithm proprietary to RSA DSI.

RC4: SymmetricCipher algorithm proprietary to RSA DSI.

---

# 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. Note: The most recent version of this document is available from JavaSoft's public Web site.

To add a new algorithm not specified herein, 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 algorithm specifications below contain the following fields:

### Name

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, and KeyPairGenerator.

### Type

The type of algorithm: Signature, MessageDigest, or KeyPairGenerator.

### Description

General notes about the algorithm, including any standards implemented by the algorithm, applicable patents, etc.

### KeyPair Algorithm (Optional)

The keypair algorithm for this algorithm.

### Strength (Optional)

For a keyed algorithm or key generation algorithm: the legal strengths for key generation or key initialization.

### Parameter Defaults (Optional)

For a key generation algorithm: the default parameter values.

### Signature format (Optional)

For a Signature algorithm, the format of the signature, that is, the input and output of the verify and sign methods, respectively.

## Algorithm Specifications

### SHA-1 Message Digest Algorithm

Name: SHA-1

Type: MessageDigest

Description: The message digest algorithm as defined in NIST's FIPS 180-1. The output of this algorithm is a 160-bit digest. Note that the term "SHA" is often used, but

it always refers to SHA-1. The first SHA, as published in FIPS 180, is obsolete. Its legal Java Cryptography Architecture name is SHA-0.

**MD2 Message Digest Algorithm**

Name: MD2

Type: MessageDigest

Description: The message digest algorithm as defined in RFC 1319. The output of this algorithm is a 128-bit (16 byte) digest.

**MD5 Message Digest Algorithm**

Name: MD5

Type: MessageDigest

Description: The message digest algorithm as defined in RFC 1321. The output of this algorithm is a 128-bit (16 byte) digest.

**The Digital Signature Algorithm**

Name: DSA

Type: Signature

Description: This algorithm is the signature algorithm described in NIST FIPS 186, using DSA with the SHA-1 message digest algorithm.

KeyPair Algorithm: DSA

Signature Format: a DER sequence of two ASN.1 INTEGER values: `r` and `s`, in that order: SEQUENCE ::= { r INTEGER, s INTEGER }

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

Names: MD2/RSA, MD5/RSA and SHA-1/RSA

Type: Signature

Description: These are the signature algorithms that use the MD2, MD5, and SHA-1 message digest algorithms (respectively) with RSA encryption.

KeyPair Algorithm: RSA

Signature Format: A 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.

**DSA KeyPair Generation Algorithm**

Name: DSA

Type: KeyPairGenerator

Description: This algorithm is the key pair generation algorithm described in NIST FIPS 186 for DSA.

Strength: The length, in bits, of the modulus `p`. This can be any integer that is a multiple of 8, greater than or equal to 512.

Parameter Defaults: The following default parameter values are used for strengths of 512, 768, and 1024 bits.

**512-bit Key Parameters**

```
SEED = b869c82b 35d70e1b 1ff91b28 e37a62ec dc34409b

counter = 123

p = fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3
    ae1617ae 01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151
    bdc43ee7 37592e17

q = 962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5

g = 678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d
    14271b9e 35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a
    6c416e50 be794ca4
```

**768-bit key parameters**

```
SEED = 77d0f8c4 dad15eb8 c4f2f8d6 726cefd9 6d5bb399

counter = 263

p = e9e64259 9d355f37 c97ffd35 67120b8e 25c9cd43 e927b3a9 670fbec5
    d8901419 22d2c3b3 ad248009 3799869d 1e846aab 49fab0ad 26d2ce6a
    22219d47 0bce7d77 7d4a21fb e9c270b5 7f607002 f3cef839 3694cf45
    ee3688c1 1a8c56ab 127a3daf

q = 9cdbd84c 9f1ac2f3 8d0f80f4 2ab952e7 338bf511

g = 30470ad5 a005fb14 ce2d9dcd 87e38bc7 d1b1c5fa cbaecbe9 5f190aa7
    a31d23c4 dbbcbe06 17454440 1a5b2c02 0965d8c2 bd2171d3 66844577
    1f74ba08 4d2029d8 3c1c1585 47f3a9f1 a2715be2 3d51ae4d 3e5a1f6a
    7064f316 933a346d 3f529252
```

**1024-bit key parameters**

```
SEED = 8d515589 4229d5e6 89ee01e6 018a237e 2cae64cd

counter = 92

p = fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80
    b6512669 455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b
    801d346f f26660b7 6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6
    1bf83b57 e7c6a8a6 150f04fb 83f6d3c5 1ec30235 54135a16 9132f675
    f3ae2b61 d72aeff2 2203199d d14801c7

q = 9760508f 15230bcc b292b982 a2eb840b f0581cf5
```

```
p = f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b
    3d078267 5159578e bad4594f e6710710 8180b449 167123e8 4c281613
    b7cf0932 8cc8a6e1 3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f
    0bfa2135 62f1fb62 7a01243b cca4f1be a8519089 a883dfe1 5ae59f06
    928b665e 807b5525 64014c3b fecf492a
```

### RSA KeyPair Generation Algorithm

Name: RSA

Type: KeyPairGenerator

Description: This algorithm is the key pair generation algorithm described in PKCS#1.

Strength: Any integer that is a multiple of 8, greater than or equal to 512.

---



Please send comments to: java-security@java.sun.com