**Java Cryprography Archietecture**

**Introduction**

The Java platform strongly emphasizes security, including language safety, cryptography, public key infrastructure, authentication, secure communication, and access control.

The JCA is a major piece of the platform, and contains a "provider" architecture and a set of APIs for digital signatures, message digests (hashes), certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), key generation and management, and secure random number generation, to name a few. These APIs allow developers to easily integrate security into their application code. The architecture was designed around the following principles:

* **Implementation independence**: Applications do not need to implement security algorithms. Rather, they can request security services from the Java platform. Security services are implemented in providers (see below), which are plugged into the Java platform via a standard interface. An application may rely on multiple independent providers for security functionality.
* **Implementation interoperability**: Providers are interoperable across applications. Specifically, an application is not bound to a specific provider, and a provider is not bound to a specific application.
* **Algorithm extensibility**: The Java platform includes a number of built-in providers that implement a basic set of security services that are widely used today. However, some applications may rely on emerging standards not yet implemented, or on proprietary services. The Java platform supports the installation of custom providers that implement such services.

Other cryptographic communication libraries available in the JDK use the JCA provider architecture, but are described elsewhere. The [Java Secure Socket Extension (JSSE)](https://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html) provides access to Secure Socket Layer (SSL) and Transport Layer Security (TLS) implementations. The [Java Generic Security Services (JGSS)](https://docs.oracle.com/javase/7/docs/technotes/guides/security/jgss/tutorials/index.html) (via Kerberos) APIs, and the [Simple Authentication and Security Layer (SASL)](https://docs.oracle.com/javase/7/docs/technotes/guides/security/sasl/sasl-refguide.html) can also be used for securely exchanging messages between communicating applications.

**Notes on Terminology**

* Prior to JDK 1.4, the JCE was an unbundled product, and as such, the JCA and JCE were regularly referred to as separate, distinct components. As JCE is now bundled in the JDK, the distinction is becoming less apparent. Since the JCE uses the same architecture as the JCA, the JCE should be more properly thought of as a part of the JCA.
* The JCA within the JDK includes two software components:
  1. the framework that defines and supports cryptographic services for which providers supply implementations. This framework includes packages such as java.security, javax.crypto, javax.crypto.spec, and javax.crypto.interfaces.
  2. the actual providers such as Sun, SunRsaSign, SunJCE, which contain the actual cryptographic implementations.

Whenever a specific JCA provider is mentioned, it will be referred to explicitly by the provider's name.

## Design Principles

The 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. While 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](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "MessageDigest), [Signature](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Signature), [KeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeyFactory), [KeyPairGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeyPairGenerator), and [Cipher](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Cipher) classes.

Implementation independence is achieved using a "provider"-based architecture. The term [Cryptographic Service Provider (CSP)](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderArch) (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 conversion 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.

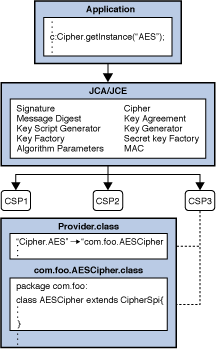
Algorithm extensibility means that new algorithms that fit in one of the supported engine classes can be added easily.

To make this clearer, review the following code and illustration:

import javax.crypto.\*;

Cipher c = Cipher.getInstance("AES");

c.init(ENCRYPT\_MODE, key);



[Description of Figure 2: Example of How Application Retrieves "AES" Cipher Instance](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "example-of-how-application-retrieves-aes-cipher-intstance)

Here an application wants an "AES" javax.crypto.Cipher instance, and doesn't care which provider is used. The application calls the getInstance() factory methods of the Cipher engine class, which in turn asks the JCA framework to find the first provider instance that supports "AES". The framework consults each installed provider, and obtains the provider's instance of the Provider class. (Recall that the Provider class is a database of available algorithms.) The framework searches each provider, finally finding a suitable entry in CSP3. This database entry points to the implementation class com.foo.AESCipher which extends CipherSpi, and is thus suitable for use by the Cipher engine class. An instance of com.foo.AESCipher is created, and is encapsulated in a newly-created instance of javax.crypto.Cipher, which is returned to the application. When the application now does the init() operation on the Cipher instance, the Cipher engine class routes the request into the corresponding engineInit() backing method in the com.foo.AESCipher class.

### Engine Classes and Algorithms

An engine class provides the interface to a specific type of cryptographic service, independent of a particular cryptographic algorithm or provider. The engines either provide:

* cryptographic operations (encryption, digital signatures, message digests, etc.),
* generators or converters of cryptographic material (keys and algorithm parameters), or
* objects (keystores or certificates) that encapsulate the cryptographic data and can be used at higher layers of abstraction.

The following engine classes are available:

* [SecureRandom](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#SecureRandom): used to generate random or pseudo-random numbers.
* [MessageDigest](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#MessageDigest): used to calculate the message digest (hash) of specified data.
* [Signature](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Signature): initialized with keys, these are used to sign data and verify digital signatures.
* [Cipher](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Cipher): initialized with keys, these are used for encrypting/decrypting data. There are various types of algorithms: symmetric bulk encryption (e.g. AES), asymmetric encryption (e.g. RSA), and password-based encryption (e.g. PBE).
* [Message Authentication Codes (MAC)](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Mac): like MessageDigests, these also generate hash values, but are first initialized with keys to protect the integrity of messages.
* [KeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyFactory): used to convert existing opaque cryptographic keys of type [Key](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Key) into key specifications (transparent representations of the underlying key material), and vice versa.
* [SecretKeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#SecretKeyFactory): used to convert existing opaque cryptographic keys of type [SecretKey](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "Key) into key specifications (transparent representations of the underlying key material), and vice versa. SecretKeyFactorys are specialized KeyFactorys that create secret (symmetric) keys only.
* [KeyPairGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyPairGenerator): used to generate a new pair of public and private keys suitable for use with a specified algorithm.
* [KeyGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyGenerator): used to generate new secret keys for use with a specified algorithm.
* [KeyAgreement](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyAgreement): used by two or more parties to agree upon and establish a specific key to use for a particular cryptographic operation.
* [AlgorithmParameters](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#AlgorithmParameters): used to store the parameters for a particular algorithm, including parameter encoding and decoding.
* [AlgorithmParameterGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#AlgorithmParameterGenerator) : used to generate a set of AlgorithmParameters suitable for a specified algorithm.
* [KeyStore](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#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.
* [CertificateFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#CertificateFactory): used to create public key certificates and Certificate Revocation Lists (CRLs).
* [CertPathBuilder](https://docs.oracle.com/javase/7/docs/technotes/guides/security/certpath/CertPathProgGuide.html): used to build certificate chains (also known as certification paths).
* [CertPathValidator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/certpath/CertPathProgGuide.html): used to validate certificate chains.
* [CertStore](https://docs.oracle.com/javase/7/docs/technotes/guides/security/certpath/CertPathProgGuide.html): 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).

**Core Classes and Interfaces**

This section discusses the core classes and interfaces provided in the JCA:

* the [Provider](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Provider) and [Security](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Security) classes,
* the [SecureRandom](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#SecureRandom), [MessageDigest](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#MessageDigest), [Signature](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Signature), [Cipher](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Cipher), [Mac](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Mac), [KeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyFactory), [SecretKeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#SecretKeyFactory), [KeyPairGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyPairGenerator), [KeyGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyGenerator), [KeyAgreement](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyAgreement), [AlgorithmParameters](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#AlgorithmParameters), [AlgorithmParameterGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#AlgorithmParameterGenerator) , [KeyStore](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyStore), and [CertificateFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "CertificateFactory), engine classes,
* the [Key](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Key) interfaces and classes,
* the [Algorithm Parameter Specification Interfaces and Classes](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#AlgSpec) and the [Key Specification Interfaces and Classes](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeySpecs), and
* miscellaneous support and convenience interfaces and classes.

### How Provider Implementations Are Requested and Supplied

For each [engine class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Engine) in the API, a implementation instance is requested and instantiated by calling one of the getInstance methods 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.

static *EngineClassName* getInstance(String algorithm)

throws NoSuchAlgorithmException

static *EngineClassName* getInstance(String algorithm, String provider)

throws NoSuchAlgorithmException, NoSuchProviderException

static *EngineClassName* getInstance(String algorithm, Provider provider)

throws NoSuchAlgorithmException

where *EngineClassName* is the desired engine type (MessageDigest/Cipher/etc). For example:

MessageDigest md = MessageDigest.getInstance("SHA-256");

KeyAgreement ka = KeyAgreement.getInstance("DH", "SunJCE");