**Java KeyStore**

The Java *KeyStore* is a database that can contain keys. A Java KeyStore is represented by the KeyStore(java.security.KeyStore) class. A KeyStore can be written to disk and read again. The KeyStore as a whole can be protected with a password, and each key entry in the KeyStore can be protected with its own password. This makes the KeyStore class a useful mechanism to handle encryption keys securely.

A KeyStore can hold the following types of keys:

* Private keys
* Public keys + certificates
* Secret keys

Private and public keys are used in asymmetric encryption. A public key can have an associated certificate. A certificate is a document that verifies the identity of the person, organization or device claiming to own the public key. A certificate is typically digitally signed by the verifying party as proof.

Secret keys are used in symmetric encryption. In many cases symmetric keys are negotiated when a secure connection is set up. Therefore you will more often be storing public and private keys in a KeyStorethan secret keys.

**Creating a KeyStore**

You can create a Java KeyStore instance by calling its getInstance() method. Here is an example of creating a KeyStore instance:

KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());

This example creates a KeyStore instance of Java's default type. It is also possible to create other types of KeyStore instance by passing a different parameter to the getInstance() method. For instance, here is an example that creates a PKCS12 type KeyStore:

KeyStore keyStore = KeyStore.getInstance("PKCS12");

**Loading the KeyStore**

Before a KeyStore instance can be used, it must be loaded. KeyStore instances are often written to disk or other kinds of storage for later use. That is why the KeyStore class assumes that you must read its data in before you can use it. However, it is possible to initialize an empty KeyStore instance with no data, as you will see later.

Loading the KeyStore data from a file or other storage is done by calling the KeyStore load() method. The load() takes two parameters:

1. An [**InputStream**](http://tutorials.jenkov.com/java-io/inputstream.html) from which to load the KeyStore data.
2. A char[] (char array) containing the KeyStore password.

Here is an example of loading a Java KeyStore:

char[] keyStorePassword = "123abc".toCharArray();

try(InputStream keyStoreData = new FileInputStream("keystore.ks")){

keyStore.load(keyStoreData, keyStorePassword);

}

This example loads the KeyStore file located in the keystore.ks file.

If you don't want to load any data into the KeyStore, just pass null for the InputStream parameter. Here is how loading an empty KeyStore looks:

keyStore3.load(null, keyStorePassword);

You must always load the KeyStore instance, either with data or with null. Otherwise the KeyStore is uninitialized, and all calls to its methods will throw an exception.

**Getting Keys**

You can get the keys of a Java KeyStore instance via its getEntry() method. A KeyStore entry is mapped to an alias which identifies the key, and is protected with a key password. Thus, to access a key you must pass the key alias and password to the getEntry() method. Here is an example of accessing a key entry in a KeyStore instance:

char[] keyPassword = "789xyz".toCharArray();

KeyStore.ProtectionParameter entryPassword =

new KeyStore.PasswordProtection(keyPassword);

KeyStore.Entry keyEntry = keyStore3.getEntry("keyAlias", entryPassword);

If you know that the key entry you want to access is a private key, you can cast the KeyStore.Entryinstance to a KeyStore.PrivateKeyEntry. Here is how that looks:

KeyStore.PrivateKeyEntry privateKeyEntry = (KeyStore.PrivateKeyEntry)

keyStore3.getEntry("keyAlias", entryPassword);

After casting to a KeyStore.PrivateKeyEntry you can access the private key, certificate and certificate chain via these methods:

* getPrivateKey()
* getCertificate()
* getCertificateChain()

**Setting Keys**

You can also set keys into a KeyStore instance. Here is an example of setting a secret key (symmetric key) into a KeyStore instance:

SecretKey secretKey = getSecretKey();

KeyStore.SecretKeyEntry secretKeyEntry = new KeyStore.SecretKeyEntry(secretKey);

keyStore3.setEntry("keyAlias2", secretKeyEntry, entryPassword);

**Storing the KeyStore**

Sometimes you may want to store a KeyStore to some storage (disk, database etc.) so you can load it again another time. You store a KeyStore by calling the store() method. Here is an example of storing a KeyStore

char[] keyStorePassword = "123abc".toCharArray();

try (FileOutputStream keyStoreOutputStream = new FileOutputStream("data/keystore.ks")) {

keyStore3.store(keyStoreOutputStream, keyStorePassword);

}

## 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 user 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

Of course, keystore files can be located as desired. In some environments, it may make sense for multiple keystores to exist. For example, in [JSSE (SSL/TLS)](https://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html), one keystore might hold a user's private keys, and another might hold certificates used to establish trust relationships.

In addition to the user's keystore, the JDK also maintains a system-wide keystore which is used to store trusted certificates from a variety of Certificate Authorities (CA's). These CA certificates can be used to help make trust decisions. For example, in SSL/TLS when the SunJSSE provider is presented with certificates from a remote peer, the default trustmanager will consult the:

* Unix: <java-home>/lib/ext/cacerts
* Windows: <java-home>\lib\ext\cacerts

file to determine if the connection is to be trusted. Instead of using the system-wide cacerts keystore, applications can set up and use their own keystores, or even use the user keystore described above.

### Keystore Implementation

The [KeyStore](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#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, JDK users can write additional security applications that use it.

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 recommended keystore implementation is "pkcs12". This is a cross-platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This standard is primarily meant for storing or transporting a user's private keys, certificates, and miscellaneous secrets. Arbitrary attributes can be associated with individual entries in a PKCS12 keystore.

The default keystore implementation type is "jks", which is specified in the following line in the java.security file:

keystore.type=jks

To have tools and other applications use a different default keystore implementation, you can change that line to specify another default type. For example, to use "pkcs12" as the default keystore implementation, change the line to:

keystore.type=pkcs12

Some applications, such as keytool, also let you override the default keystore type (via the -storetype command-line parameter).

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

**"jceks"**is an alternate proprietary keystore format to "jks" that uses Password-Based Encryption with Triple-DES.

The "jceks" implementation can parse and convert a "jks" keystore file to the "jceks" format. You may upgrade your keystore of type "jks" to a keystore of type "jceks" by changing the password of a private-key entry in your keystore and specifying -storetype jceks as the keystore type. To apply the cryptographically strong(er) key protection supplied to a private key named "signkey" in your default keystore, use the following command, which will prompt you for the old and new key passwords:

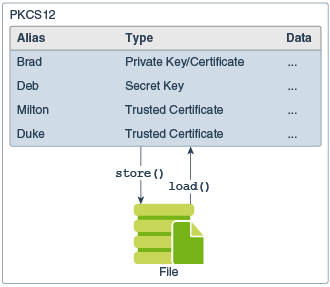
keytool -keypasswd -alias signkey -storetype jceks

See [Security Tools](https://docs.oracle.com/javase/7/docs/technotes/tools/index.html#security) for more information about keytool and about keystores and how they are managed.

Keystore implementations are provider-based. Developers interested in writing their own KeyStore implementations should consult [How to Implement a Provider for the Java Cryptography Architecture](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/HowToImplAProvider.html) for more information on this topic.

### The KeyStore Class

The KeyStore class is an [engine class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Engine) that supplies well-defined interfaces to access and modify the information in a keystore.



[Description of Figure 15: The KeyStore Class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "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

KeyStore objects are obtained by using one of the KeyStore [getInstance() static factory methods](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderImplReq).

### 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:

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:

final Enumeration aliases()

### Determining Keystore Entry Types

As stated in [The KeyStore Class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyStore), 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:

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:

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:

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 PKCS8 standard. In the other method, thepassword is the password used to protect the key.

The deleteEntry method deletes an entry:

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:

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

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

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 certificate via the following:

final String getCertificateAlias(Certificate cert)

### Saving the KeyStore

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

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.