## NAME

keytool – a key and certificate management utility

## SYNOPSIS

`keytool` [*commands*]

*commands*

Commands for `keytool` include the following:

- `-certreq`: Generates a certificate request
- `-changealias`: Changes an entry's alias
- `-delete`: Deletes an entry
- `-exportcert`: Exports certificate
- `-genkeypair`: Generates a key pair
- `-genseckey`: Generates a secret key
- `-gencert`: Generates a certificate from a certificate request
- `-importcert`: Imports a certificate or a certificate chain
- `-importpass`: Imports a password
- `-importkeystore`: Imports one or all entries from another keystore
- `-keypasswd`: Changes the key password of an entry
- `-list`: Lists entries in a keystore
- `-printcert`: Prints the content of a certificate
- `-printcertreq`: Prints the content of a certificate request
- `-printcrl`: Prints the content of a Certificate Revocation List (CRL) file
- `-storepasswd`: Changes the store password of a keystore
- `-showinfo`: Displays security–related information
- `-version`: Prints the program version

See **Commands and Options** for a description of these commands with their options.

## DESCRIPTION

The `keytool` command is a key and certificate management utility. It enables users to administer their own public/private key pairs and associated certificates for use in self–authentication (where a user authenticates themselves to other users and services) or data integrity and authentication services, by using digital signatures. The `keytool` command also enables users to cache the public keys (in the form of certificates) of their communicating peers.

A certificate is a digitally signed statement from one entity (person, company, and so on), which says that the public key (and some other information) of some other entity has a particular value. When data is digitally signed, the signature can be verified to check the data integrity and authenticity. Integrity means that the data hasn't been modified or tampered with, and authenticity means that the data comes from the individual who claims to have created and signed it.

The `keytool` command also enables users to administer secret keys and passphrases used in symmetric encryption and decryption (Data Encryption Standard). It can also display other security–related information.

The `keytool` command stores the keys and certificates in a keystore.

The `keytool` command uses the `jdk.certpath.disabledAlgorithms` and `jdk.security.legacyAlgorithms` security properties to determine which algorithms are considered a security risk. It emits warnings when disabled or legacy algorithms are being used. The `jdk.certpath.disabledAlgorithms` and `jdk.security.legacyAlgorithms` security properties are defined in the

`java.security` file (located in the JDK's `$JAVA_HOME/conf/security` directory).

## COMMAND AND OPTION NOTES

The following notes apply to the descriptions in **Commands and Options**:

- All command and option names are preceded by a hyphen sign (`-`).
- Only one command can be provided.
- Options for each command can be provided in any order.
- There are two kinds of options, one is single−valued which should be only provided once. If a single−valued option is provided multiple times, the value of the last one is used. The other type is multi−valued, which can be provided multiple times and all values are used. The only multi−valued option currently supported is the `-ext` option used to generate X.509v3 certificate extensions.
- All items not italicized or in braces ({ }) or brackets ([ ]) are required to appear as is.
- Braces surrounding an option signify that a default value is used when the option isn't specified on the command line. Braces are also used around the `-v`, `-rfc`, and `-J` options, which have meaning only when they appear on the command line. They don't have any default values.
- Brackets surrounding an option signify that the user is prompted for the values when the option isn't specified on the command line. For the `-keypass` option, if you don't specify the option on the command line, then the `keytool` command first attempts to use the keystore password to recover the private/secret key. If this attempt fails, then the `keytool` command prompts you for the private/secret key password.
- Items in italics (option values) represent the actual values that must be supplied. For example, here is the format of the `-printcert` command:

  `keytool -printcert` {`-file` *cert_file*} {`-v`}

  When you specify a `-printcert` command, replace *cert_file* with the actual file name, as follows: `keytool -printcert -file VScert.cer`
- Option values must be enclosed in quotation marks when they contain a blank (space).

## COMMANDS AND OPTIONS

The keytool commands and their options can be grouped by the tasks that they perform.

**Commands for Creating or Adding Data to the Keystore**:

- `-gencert`
- `-genkeypair`
- `-genseckey`
- `-importcert`
- `-importpass`

**Commands for Importing Contents from Another Keystore**:

- `-importkeystore`

**Commands for Generating a Certificate Request**:

- `-certreq`

**Commands for Exporting Data**:

- `-exportcert`

**Commands for Displaying Data**:

- `-list`

- `-printcert`
- `-printcertreq`
- `-printcrl`

**Commands for Managing the Keystore**:

- `-storepasswd`
- `-keypasswd`
- `-delete`
- `-changealias`

**Commands for Displaying Security−related Information**:

- `-showinfo`

**Commands for Displaying Program Version**:

- `-version`

## COMMANDS FOR CREATING OR ADDING DATA TO THE KEYSTORE

`-gencert`

The following are the available options for the `-gencert` command:

- {`-rfc`}: Output in RFC (Request For Comment) style
- {`-infile` *infile*}: Input file name
- {`-outfile` *outfile*}: Output file name
- {`-alias` *alias*}: Alias name of the entry to process
- {`-sigalg` *sigalg*}: Signature algorithm name
- {`-dname` *dname*}: Distinguished name
- {`-startdate` *startdate*}: Certificate validity start date and time
- {`-ext` *ext*}*: X.509 extension
- {`-validity` *days*}: Validity number of days
- [`-keypass` *arg*]: Key password
- {`-keystore` *keystore*}: Keystore name
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Adds a security provider by name (such as SunPKCS11) with an optional configure argument. The value of the security provider is the name of a security provider that is defined in a module.

  For example,

  ```
  keytool -addprovider SunPKCS11 -providerarg some.cfg ...
  ```

  **Note:**

  For compatibility reasons, the SunPKCS11 provider can still be loaded with `-providerclass sun.security.pkcs11.SunPKCS11` even if it is now defined in a module. This is the only module included in the JDK that needs a configuration, and therefore the most widely used with the `-providerclass` option. For legacy security providers located on classpath and loaded by reflection, `-providerclass` should still be used.

- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.

  For example, if `MyProvider` is a legacy provider loaded via reflection,

  ```
  keytool -providerclass com.example.MyProvider ...
  ```

- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output
- {`-protected`}: Password provided through a protected mechanism

Use the `-gencert` command to generate a certificate as a response to a certificate request file (which can be created by the `keytool -certreq` command). The command reads the request either from *infile* or, if omitted, from the standard input, signs it by using the alias's private key, and outputs the X.509 certificate into either *outfile* or, if omitted, to the standard output. When `-rfc` is specified, the output format is Base64–encoded PEM; otherwise, a binary DER is created.

The `-sigalg` value specifies the algorithm that should be used to sign the certificate. The *startdate* argument is the start time and date that the certificate is valid. The *days* argument tells the number of days for which the certificate should be considered valid.

When *dname* is provided, it is used as the subject of the generated certificate. Otherwise, the one from the certificate request is used.

The `-ext` value shows what X.509 extensions will be embedded in the certificate. Read **Common Command Options** for the grammar of `-ext`.

The `-gencert` option enables you to create certificate chains. The following example creates a certificate, `e1`, that contains three certificates in its certificate chain.

The following commands creates four key pairs named `ca`, `ca1`, `ca2`, and `e1`:

```
keytool -alias ca -dname CN=CA -genkeypair -keyalg rsa
keytool -alias ca1 -dname CN=CA -genkeypair -keyalg rsa
keytool -alias ca2 -dname CN=CA -genkeypair -keyalg rsa
keytool -alias e1 -dname CN=E1 -genkeypair -keyalg rsa
```

The following two commands create a chain of signed certificates; `ca` signs `ca1` and `ca1` signs `ca2`, all of which are self–issued:

```
keytool -alias ca1 -certreq |
    keytool -alias ca -gencert -ext san=dns:ca1 |
    keytool -alias ca1 -importcert

keytool -alias ca2 -certreq |
    keytool -alias ca1 -gencert -ext san=dns:ca2 |
    keytool -alias ca2 -importcert
```

The following command creates the certificate `e1` and stores it in the `e1.cert` file, which is signed by `ca2`. As a result, `e1` should contain `ca`, `ca1`, and `ca2` in its certificate chain:

```
keytool -alias e1 -certreq | keytool -alias ca2 -gencert >
e1.cert
```

`-genkeypair`

The following are the available options for the `-genkeypair` command:

- {`-alias` *alias*}: Alias name of the entry to process
- `-keyalg` *alg*: Key algorithm name
- {`-keysize` *size*}: Key bit size

- {`-groupname` *name*}: Group name. For example, an Elliptic Curve name.
- {`-sigalg` *alg*}: Signature algorithm name
- {`-signer` *alias*}: Signer alias
- [`-signerkeypass` *arg*]: Signer key password
- [`-dname` *name*]: Distinguished name
- {`-startdate` *date*}: Certificate validity start date and time
- {`-ext` *value*}*: X.509 extension
- {`-validity` *days*}: Validity number of days
- [`-keypass` *arg*]: Key password
- {`-keystore` *keystore*}: Keystore name
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*] }: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output
- {`-protected`}: Password provided through a protected mechanism

Use the `-genkeypair` command to generate a key pair (a public key and associated private key). When the `-signer` option is not specified, the public key is wrapped in an X.509 v3 self−signed certificate and stored as a single−element certificate chain. When the `-signer` option is specified, a new certificate is generated and signed by the designated signer and stored as a multiple−element certificate chain (containing the generated certificate itself, and the signer's certificate chain). The certificate chain and private key are stored in a new keystore entry that is identified by its alias.

The `-keyalg` value specifies the algorithm to be used to generate the key pair. The `-keysize` value specifies the size of each key to be generated. The `-groupname` value specifies the named group (for example, the standard or predefined name of an Elliptic Curve) of the key to be generated.

When a `-keysize` value is provided, it will be used to initialize a `KeyPairGenerator` object using the `initialize(int keysize)` method. When a `-groupname` value is provided, it will be used to initialize a `KeyPairGenerator` object using the `initialize(Algorithm−ParameterSpec params)` method where `params` is `new NamedParameter−Spec(groupname)`.

Only one of `-groupname` and `-keysize` can be specified. If an algorithm has multiple named groups that have the same key size, the `-groupname` option should usually be used. In this case, if `-keysize` is specified, it's up to the security provider to determine which named group is chosen when generating a key pair.

The `-sigalg` value specifies the algorithm that should be used to sign the certificate. This algorithm must be compatible with the `-keyalg` value.

The `-signer` value specifies the alias of a `PrivateKeyEntry` for the signer that already exists in the keystore. This option is used to sign the certificate with the signer's private key. This is especially useful for key agreement algorithms (i.e. the `-keyalg` value is `XDH`, `X25519`, `X448`,

or `DH`) as these keys cannot be used for digital signatures, and therefore a self−signed certificate cannot be created.

The `−signerkeypass` value specifies the password of the signer's private key. It can be specified if the private key of the signer entry is protected by a password different from the store password.

The `−dname` value specifies the X.500 Distinguished Name to be associated with the value of `−alias`. If the `−signer` option is not specified, the issuer and subject fields of the self−signed certificate are populated with the specified distinguished name. If the `−signer` option is specified, the subject field of the certificate is populated with the specified distinguished name and the issuer field is populated with the subject field of the signer's certificate. If a distinguished name is not provided at the command line, then the user is prompted for one.

The value of `−keypass` is a password used to protect the private key of the generated key pair. If a password is not provided, then the user is prompted for it. If you press the **Return** key at the prompt, then the key password is set to the same password as the keystore password. The `−keypass` value must have at least six characters.

The value of `−startdate` specifies the issue time of the certificate, also known as the "Not Before" value of the X.509 certificate's Validity field.

The option value can be set in one of these two forms:

([+−]*nnn*[`ymdHMS`])+

[*yyyy*/*mm*/*dd*] [*HH*:*MM*:*SS*]

With the first form, the issue time is shifted by the specified value from the current time. The value is a concatenation of a sequence of subvalues. Inside each subvalue, the plus sign (+) means shift forward, and the minus sign (−) means shift backward. The time to be shifted is *nnn* units of years, months, days, hours, minutes, or seconds (denoted by a single character of `y`, `m`, `d`, `H`, `M`, or `S` respectively). The exact value of the issue time is calculated by using the `java.util.GregorianCalendar.add(int field, int amount)` method on each subvalue, from left to right. For example, the issue time can be specified by:

```
Calendar c = new GregorianCalendar();
c.add(Calendar.YEAR, -1);
c.add(Calendar.MONTH, 1);
c.add(Calendar.DATE, -1);
return c.getTime()
```

With the second form, the user sets the exact issue time in two parts, year/month/day and hour:minute:second (using the local time zone). The user can provide only one part, which means the other part is the same as the current date (or time). The user must provide the exact number of digits shown in the format definition (padding with 0 when shorter). When both date and time are provided, there is one (and only one) space character between the two parts. The hour should always be provided in 24−hour format.

When the option isn't provided, the start date is the current time. The option can only be provided one time.

The value of *date* specifies the number of days (starting at the date specified by `−startdate`, or the current date when `−startdate` isn't specified) for which the certificate should be considered valid.

`−genseckey`

The following are the available options for the `−genseckey` command:

- {`−alias` *alias*}: Alias name of the entry to process

- [`-keypass` *arg*]: Key password
- `-keyalg` *alg*: Key algorithm name
- {`-keysize` *size*}: Key bit size
- {`-keystore` *keystore*}: Keystore name
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output
- {`-protected`}: Password provided through a protected mechanism

Use the `-genseckey` command to generate a secret key and store it in a new `KeyStore.SecretKeyEntry` identified by `alias`.

The value of `-keyalg` specifies the algorithm to be used to generate the secret key, and the value of `-keysize` specifies the size of the key that is generated. The `-keypass` value is a password that protects the secret key. If a password is not provided, then the user is prompted for it. If you press the **Return** key at the prompt, then the key password is set to the same password that is used for the `-keystore`. The `-keypass` value must contain at least six characters.

`-importcert`

The following are the available options for the `-importcert` command:

- {`-noprompt`}: Do not prompt
- {`-trustcacerts`}: Trust certificates from cacerts
- {`-protected`}: Password is provided through protected mechanism
- {`-alias` *alias*}: Alias name of the entry to process
- {`-file` *file*}: Input file name
- [`-keypass` *arg*]: Key password
- {`-keystore` *keystore*}: Keystore name
- {`-cacerts`}: Access the cacerts keystore
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output

Use the `-importcert` command to read the certificate or certificate chain (where the latter is supplied in a PKCS#7 formatted reply or in a sequence of X.509 certificates) from `-file` *file*,

and store it in the `keystore` entry identified by `-alias`. If `-file` *file* is not specified, then the certificate or certificate chain is read from `stdin`.

The `keytool` command can import X.509 v1, v2, and v3 certificates, and PKCS#7 formatted certificate chains consisting of certificates of that type. The data to be imported must be provided either in binary encoding format or in printable encoding format (also known as Base64 encoding) as defined by the Internet RFC 1421 standard. In the latter case, the encoding must be bounded at the beginning by a string that starts with `-----BEGIN`, and bounded at the end by a string that starts with `-----END`.

You import a certificate for two reasons: To add it to the list of trusted certificates, and to import a certificate reply received from a certificate authority (CA) as the result of submitting a Certificate Signing Request (CSR) to that CA. See the `-certreq` command in **Commands for Generating a Certificate Request**.

The type of import is indicated by the value of the `-alias` option. If the alias doesn't point to a key entry, then the `keytool` command assumes you are adding a trusted certificate entry. In this case, the alias shouldn't already exist in the keystore. If the alias does exist, then the `keytool` command outputs an error because a trusted certificate already exists for that alias, and doesn't import the certificate. If `-alias` points to a key entry, then the `keytool` command assumes that you're importing a certificate reply.

`-importpass`

The following are the available options for the `-importpass` command:

- {`-alias` *alias*}: Alias name of the entry to process
- [`-keypass` *arg*]: Key password
- {`-keyalg` *alg*}: Key algorithm name
- {`-keysize` *size*}: Key bit size
- {`-keystore` *keystore*}: Keystore name
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output
- {`-protected`}: Password provided through a protected mechanism

Use the `-importpass` command to imports a passphrase and store it in a new `KeyStore.SecretKeyEntry` identified by `-alias`. The passphrase may be supplied via the standard input stream; otherwise the user is prompted for it. The `-keypass` option provides a password to protect the imported passphrase. If a password is not provided, then the user is prompted for it. If you press the **Return** key at the prompt, then the key password is set to the same password as that used for the `keystore`. The `-keypass` value must contain at least six characters.

## COMMANDS FOR IMPORTING CONTENTS FROM ANOTHER KEYSTORE

`-importkeystore`

The following are the available options for the `-importkeystore` command:

- `-srckeystore` *keystore*: Source keystore name
- {`-destkeystore` *keystore*}: Destination keystore name
- {`-srcstoretype` *type*}: Source keystore type
- {`-deststoretype` *type*}: Destination keystore type
- [`-srcstorepass` *arg*]: Source keystore password
- [`-deststorepass` *arg*]: Destination keystore password
- {`-srcprotected`}: Source keystore password protected
- {`-destprotected`}: Destination keystore password protected
- {`-srcprovidername` *name*}: Source keystore provider name
- {`-destprovidername` *name*}: Destination keystore provider name
- {`-srcalias` *alias*}: Source alias
- {`-destalias` *alias*}: Destination alias
- [`-srckeypass` *arg*]: Source key password
- [`-destkeypass` *arg*]: Destination key password
- {`-noprompt`}: Do not prompt
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output

**Note:**

This is the first line of all options:

`-srckeystore` *keystore* `-destkeystore` *keystore*

Use the `-importkeystore` command to import a single entry or all entries from a source keystore to a destination keystore.

**Note:**

If you do not specify `-destkeystore` when using the `keytool -importkeystore` command, then the default keystore used is `$HOME/.keystore`.

When the `-srcalias` option is provided, the command imports the single entry identified by the alias to the destination keystore. If a destination alias isn't provided with `-destalias`, then `-srcalias` is used as the destination alias. If the source entry is protected by a password, then `-srckeypass` is used to recover the entry. If `-srckeypass` isn't provided, then the `keytool` command attempts to use `-srcstorepass` to recover the entry. If `-srcstorepass` is not provided or is incorrect, then the user is prompted for a password. The destination entry is protected with `-destkeypass`. If `-destkeypass` isn't provided, then the destination entry is protected with the source entry password. For example, most third–party tools require `storepass` and `keypass` in a PKCS #12 keystore to be the same. To create a PKCS#12 keystore for these tools, always specify a `-destkeypass` that is the same as `-deststorepass`.

If the `-srcalias` option isn't provided, then all entries in the source keystore are imported into the destination keystore. Each destination entry is stored under the alias from the source entry. If the source entry is protected by a password, then `-srcstorepass` is used to recover the entry. If `-srcstorepass` is not provided or is incorrect, then the user is prompted for a password. If a source keystore entry type isn't supported in the destination keystore, or if an error occurs while

storing an entry into the destination keystore, then the user is prompted either to skip the entry and continue or to quit. The destination entry is protected with the source entry password.

If the destination alias already exists in the destination keystore, then the user is prompted either to overwrite the entry or to create a new entry under a different alias name.

If the `-noprompt` option is provided, then the user isn't prompted for a new destination alias. Existing entries are overwritten with the destination alias name. Entries that can't be imported are skipped and a warning is displayed.

## COMMANDS FOR GENERATING A CERTIFICATE REQUEST

`-certreq`

The following are the available options for the `-certreq` command:

- {`-alias` *alias*}: Alias name of the entry to process
- {`-sigalg` *alg*}: Signature algorithm name
- {`-file` *file*}: Output file name
- [ `-keypass` *arg*]: Key password
- {`-keystore` *keystore*}: Keystore name
- {`-dname` *name*}: Distinguished name
- {`-ext` *value*}: X.509 extension
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output
- {`-protected`}: Password provided through a protected mechanism

Use the `-certreq` command to generate a Certificate Signing Request (CSR) using the PKCS #10 format.

A CSR is intended to be sent to a CA. The CA authenticates the certificate requestor (usually offline) and returns a certificate or certificate chain to replace the existing certificate chain (initially a self–signed certificate) in the keystore.

The private key associated with *alias* is used to create the PKCS #10 certificate request. To access the private key, the correct password must be provided. If `-keypass` isn't provided at the command line and is different from the password used to protect the integrity of the keystore, then the user is prompted for it. If `-dname` is provided, then it is used as the subject in the CSR. Otherwise, the X.500 Distinguished Name associated with alias is used.

The `-sigalg` value specifies the algorithm that should be used to sign the CSR.

The CSR is stored in the `-file` *file*. If a file is not specified, then the CSR is output to `-stdout`.

Use the `-importcert` command to import the response from the CA.

## COMMANDS FOR EXPORTING DATA

`-exportcert`

The following are the available options for the `-exportcert` command:

- {`-rfc`}: Output in RFC style
- {`-alias` *alias*}: Alias name of the entry to process
- {`-file` *file*}: Output file name
- {`-keystore` *keystore*}: Keystore name
- {`-cacerts`}: Access the cacerts keystore
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*] }: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output
- {`-protected`}: Password provided through a protected mechanism

Use the `-exportcert` command to read a certificate from the keystore that is associated with `-alias` *alias* and store it in the `-file` *file*. When a file is not specified, the certificate is output to `stdout`.

By default, the certificate is output in binary encoding. If the `-rfc` option is specified, then the output in the printable encoding format defined by the Internet RFC 1421 Certificate Encoding Standard.

If `-alias` refers to a trusted certificate, then that certificate is output. Otherwise, `-alias` refers to a key entry with an associated certificate chain. In that case, the first certificate in the chain is returned. This certificate authenticates the public key of the entity addressed by `-alias`.

## COMMANDS FOR DISPLAYING DATA

`-list` The following are the available options for the `-list` command:

- {`-rfc`}: Output in RFC style
- {`-alias` *alias*}: Alias name of the entry to process
- {`-keystore` *keystore*}: Keystore name
- {`-cacerts`}: Access the cacerts keystore
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*] }: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output

- {`-protected`}: Password provided through a protected mechanism

Use the `-list` command to print the contents of the keystore entry identified by `-alias` to `stdout`. If `-alias` *alias* is not specified, then the contents of the entire keystore are printed.

By default, this command prints the SHA−256 fingerprint of a certificate. If the `-v` option is specified, then the certificate is printed in human−readable format, with additional information such as the owner, issuer, serial number, and any extensions. If the `-rfc` option is specified, then the certificate contents are printed by using the printable encoding format, as defined by the Internet RFC 1421 Certificate Encoding Standard.

**Note:**

You can't specify both `-v` and `-rfc` in the same command. Otherwise, an error is reported.

`-printcert`

The following are the available options for the `-printcert` command:

- {`-rfc`}: Output in RFC style
- {`-file` *cert_file*}: Input file name
- {`-sslserver` *server*[`:`*port*]}:: Secure Sockets Layer (SSL) server host and port
- {`-jarfile` *JAR_file*}: Signed `.jar` file
- {`-keystore` *keystore*}: Keystore name
- {`-trustcacerts`}: Trust certificates from cacerts
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-protected`}: Password is provided through protected mechanism
- {`-v`}: Verbose output

Use the `-printcert` command to read and print the certificate from `-file` *cert_file*, the SSL server located at `-sslserver` *server*[`:`*port*], or the signed JAR file specified by `-jarfile` *JAR_file*. It prints its contents in a human−readable format. When a port is not specified, the standard HTTPS port 443 is assumed.

**Note:**

The `-sslserver` and `-file` options can't be provided in the same command. Otherwise, an error is reported. If you don't specify either option, then the certificate is read from `stdin`.

When`-rfc` is specified, the `keytool` command prints the certificate in PEM mode as defined by the Internet RFC 1421 Certificate Encoding standard.

If the certificate is read from a file or `stdin`, then it might be either binary encoded or in printable encoding format, as defined by the RFC 1421 Certificate Encoding standard.

If the SSL server is behind a firewall, then the `-J-Dhttps.proxyHost=proxyhost` and `-J-Dhttps.proxyPort=proxyport` options can be specified on the command line for proxy tunneling.

**Note:**

This command can be used independently of a keystore. This command does not check for the

weakness of a certificate's signature algorithm if it is a trusted certificate in the user keystore (specified by `-keystore`) or in the `cacerts` keystore (if `-trustcacerts` is specified).

`-printcertreq`

The following are the available options for the `-printcertreq` command:

- {`-file` *file*}: Input file name
- {`-v`}: Verbose output

Use the `-printcertreq` command to print the contents of a PKCS #10 format certificate request, which can be generated by the `keytool -certreq` command. The command reads the request from file. If there is no file, then the request is read from the standard input.

`-printcrl`

The following are the available options for the `-printcrl` command:

- {`-file crl`}: Input file name
- {`-keystore` *keystore*}: Keystore name
- {`-trustcacerts`}: Trust certificates from cacerts
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-protected`}: Password is provided through protected mechanism
- {`-v`}: Verbose output

Use the `-printcrl` command to read the Certificate Revocation List (CRL) from `-file crl` . A CRL is a list of the digital certificates that were revoked by the CA that issued them. The CA generates the `crl` file.

**Note:**

This command can be used independently of a keystore. This command attempts to verify the CRL using a certificate from the user keystore (specified by `-keystore`) or the `cacerts` keystore (if `-trustcacerts` is specified), and will print out a warning if it cannot be verified.

## COMMANDS FOR MANAGING THE KEYSTORE

`-storepasswd`

The following are the available options for the `-storepasswd` command:

- [`-new` *arg*]: New password
- {`-keystore` *keystore*}: Keystore name
- {`-cacerts`}: Access the cacerts keystore
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.

- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output

Use the `-storepasswd` command to change the password used to protect the integrity of the keystore contents. The new password is set by `-new` *arg* and must contain at least six characters.

`-keypasswd`

The following are the available options for the `-keypasswd` command:

- {`-alias` *alias*}: Alias name of the entry to process
- [`-keypass` *old_keypass*]: Key password
- [`-new` *new_keypass*]: New password
- {`-keystore` *keystore*}: Keystore name
- {`-storepass` *arg*}: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output

Use the `-keypasswd` command to change the password (under which private/secret keys identified by `-alias` are protected) from `-keypass` *old_keypass* to `-new` *new_keypass*. The password value must contain at least six characters.

If the `-keypass` option isn't provided at the command line and the `-keypass` password is different from the keystore password (`-storepass` *arg*), then the user is prompted for it.

If the `-new` option isn't provided at the command line, then the user is prompted for it.

`-delete`

The following are the available options for the `-delete` command:

- [`-alias` *alias*]: Alias name of the entry to process
- {`-keystore` *keystore*}: Keystore name
- {`-cacerts`}: Access the cacerts keystore
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output

- {`-protected`}: Password provided through a protected mechanism

Use the `-delete` command to delete the `-alias` *alias* entry from the keystore. When not provided at the command line, the user is prompted for the `alias`.

`-changealias`

The following are the available options for the `-changealias` command:

- {`-alias` *alias*}: Alias name of the entry to process
- [`-destalias` *alias*]: Destination alias
- [`-keypass` *arg*]: Key password
- {`-keystore` *keystore*}: Keystore name
- {`-cacerts`}: Access the cacerts keystore
- [`-storepass` *arg*]: Keystore password
- {`-storetype` *type*}: Keystore type
- {`-providername` *name*}: Provider name
- {`-addprovider` *name* [`-providerarg` *arg*]}: Add security provider by name (such as SunPKCS11) with an optional configure argument.
- {`-providerclass` *class* [`-providerarg` *arg*]}: Add security provider by fully qualified class name with an optional configure argument.
- {`-providerpath` *list*}: Provider classpath
- {`-v`}: Verbose output
- {`-protected`}: Password provided through a protected mechanism

Use the `-changealias` command to move an existing keystore entry from `-alias` *alias* to a new `-destalias` *alias*. If a destination alias is not provided, then the command prompts you for one. If the original entry is protected with an entry password, then the password can be supplied with the `-keypass` option. If a key password is not provided, then the `-storepass` (if provided) is attempted first. If the attempt fails, then the user is prompted for a password.

## COMMANDS FOR DISPLAYING SECURITY–RELATED INFORMATION

`-showinfo`

The following are the available options for the `-showinfo` command:

- {`-tls`}: Displays TLS configuration information
- {`-v`}: Verbose output

Use the `-showinfo` command to display various security–related information. The `-tls` option displays TLS configurations, such as the list of enabled protocols and cipher suites.

## COMMANDS FOR DISPLAYING PROGRAM VERSION

You can use `-version` to print the program version of `keytool`.

## COMMANDS FOR DISPLAYING HELP INFORMATION

You can use `--help` to display a list of `keytool` commands or to display help information about a specific `keytool` command.

- To display a list of `keytool` commands, enter:

```
keytool --help
```

- To display help information about a specific `keytool` command, enter:

```
keytool -<command> --help
```

## COMMON COMMAND OPTIONS

The `-v` option can appear for all commands except `--help`. When the `-v` option appears, it signifies verbose mode, which means that more information is provided in the output.

The `-J`*option* argument can appear for any command. When the `-J`*option* is used, the specified *option* string is passed directly to the Java interpreter. This option doesn't contain any spaces. It's useful for adjusting the execution environment or memory usage. For a list of possible interpreter options, enter `java -h` or `java -X` at the command line.

These options can appear for all commands operating on a keystore:

`-storetype` *storetype*
: This qualifier specifies the type of keystore to be instantiated.

`-keystore` *keystore*
: The keystore location.

  If the JKS `storetype` is used and a keystore file doesn't yet exist, then certain `keytool` commands can result in a new keystore file being created. For example, if `keytool -genkeypair` is called and the `-keystore` option isn't specified, the default keystore file named `.keystore` is created in the user's home directory if it doesn't already exist. Similarly, if the `-keystore ks_file` option is specified but `ks_file` doesn't exist, then it is created. For more information on the JKS `storetype`, see the **KeyStore Implementation** section in **KeyStore aliases**.

  Note that the input stream from the `-keystore` option is passed to the `KeyStore.load` method. If `NONE` is specified as the URL, then a null stream is passed to the `KeyStore.load` method. `NONE` should be specified if the keystore isn't file–based. For example, when the keystore resides on a hardware token device.

`-cacerts` *cacerts*
: Operates on the *cacerts* keystore . This option is equivalent to `-keystore` *path_to_cacerts* `-storetype` *type_of_cacerts*. An error is reported if the `-keystore` or `-storetype` option is used with the `-cacerts` option.

`-storepass` [`:env` | `:file` ] *argument*
: The password that is used to protect the integrity of the keystore.

  If the modifier `env` or `file` isn't specified, then the password has the value *argument*, which must contain at least six characters. Otherwise, the password is retrieved as follows:

  - `env`: Retrieve the password from the environment variable named *argument*.
  - `file`: Retrieve the password from the file named *argument*.

  **Note:** All other options that require passwords, such as `-keypass`, `-srckeypass`, `-destkeypass`, `-srcstorepass`, and `-deststorepass`, accept the `env` and `file` modifiers. Remember to separate the password option and the modifier with a colon (:).

  The password must be provided to all commands that access the keystore contents. For such commands, when the `-storepass` option isn't provided at the command line, the user is prompted for it.

  When retrieving information from the keystore, the password is optional. If a password is not specified, then the integrity of the retrieved information can't be verified and a warning is displayed.

`-providername` *name*
: Used to identify a cryptographic service provider's name when listed in the security properties file.

`-addprovider` *name*
: Used to add a security provider by name (such as SunPKCS11) .

`-providerclass` *class*
: Used to specify the name of a cryptographic service provider's master class file when the service provider isn't listed in the security properties file.

`-providerpath` *list*
: Used to specify the provider classpath.

`-providerarg` *arg*
: Used with the `-addprovider` or `-providerclass` option to represent an optional string input argument for the constructor of *class* name.

`-protected=true|false`
: Specify this value as `true` when a password must be specified by way of a protected authentication path, such as a dedicated PIN reader. Because there are two keystores involved in the `-importkeystore` command, the following two options, `-srcprotected` and `-destprotected`, are provided for the source keystore and the destination keystore respectively.

`-ext` {*name*{`:critical`} {=*value*}}
: Denotes an X.509 certificate extension. The option can be used in `-genkeypair` and `-gencert` to embed extensions into the generated certificate, or in `-certreq` to show what extensions are requested in the certificate request. The option can appear multiple times. The *name* argument can be a supported extension name (see **Supported Named Extensions**) or an arbitrary OID number. The *value* argument, when provided, denotes the argument for the extension. When *value* is omitted, the default value of the extension or the extension itself requires no argument. The `:critical` modifier, when provided, means the extension's `isCritical` attribute is `true`; otherwise, it is `false`. You can use `:c` in place of `:critical`.

`-conf` *file*
: Specifies a pre-configured options file.

## PRE-CONFIGURED OPTIONS FILE

A pre-configured options file is a Java properties file that can be specified with the `-conf` option. Each property represents the default option(s) for a keytool command using "keytool.*command_name*" as the property name. A special property named "keytool.all" represents the default option(s) applied to all commands. A property value can include `${prop}` which will be expanded to the system property associated with it. If an option value includes white spaces inside, it should be surrounded by quotation marks (" or '). All property names must be in lower case.

When `keytool` is launched with a pre-configured options file, the value for "keytool.all" (if it exists) is prepended to the `keytool` command line first, with the value for the command name (if it exists) comes next, and the existing options on the command line at last. For a single-valued option, this allows the property for a specific command to override the "keytool.all" value, and the value specified on the command line to override both. For multiple-valued options, all of them will be used by `keytool`.

For example, given the following file named `preconfig`:

```
# A tiny pre-configured options file
keytool.all = -keystore ${user.home}/ks
keytool.list = -v
keytool.genkeypair = -keyalg rsa
```

`keytool -conf preconfig -list` is identical to

```
keytool -keystore ~/ks -v -list
```

`keytool -conf preconfig -genkeypair -alias me` is identical to

```
keytool -keystore ~/ks -keyalg rsa -genkeypair -alias me
```

`keytool -conf preconfig -genkeypair -alias you -keyalg ec` is identical to

```
keytool -keystore ~/ks -keyalg rsa -genkeypair -alias you -keyalg
ec
```

which is equivalent to

```
keytool -keystore ~/ks -genkeypair -alias you -keyalg ec
```

because `-keyalg` is a single-valued option and the `ec` value specified on the command line overrides the preconfigured options file.

## EXAMPLES OF OPTION VALUES

The following examples show the defaults for various option values:

```
-alias "mykey"

-keysize
    2048 (when using -genkeypair and -keyalg is "DSA")
    3072 (when using -genkeypair and -keyalg is "RSA", "RSASSA-PSS", or "DH
    384 (when using -genkeypair and -keyalg is "EC")
    56 (when using -genseckey and -keyalg is "DES")
    168 (when using -genseckey and -keyalg is "DESede")

-groupname
    ed25519 (when using -genkeypair and -keyalg is "EdDSA", key size is 255
    x25519 (when using -genkeypair and -keyalg is "XDH", key size is 255)

-validity 90

-keystore <the file named .keystore in the user's home directory>

-destkeystore <the file named .keystore in the user's home directory>

-storetype <the value of the "keystore.type" property in the
    security properties file, which is returned by the static
    getDefaultType method in java.security.KeyStore>

-file
    stdin (if reading)
    stdout (if writing)

-protected false
```

When generating a certificate or a certificate request, the default signature algorithm (`-sigalg` option) is derived from the algorithm of the underlying private key to provide an appropriate level of security strength as follows:

Default Signature Algorithms

| keyalg | key size | default sigalg |
|---|---|---|
| DSA | any size | SHA256withDSA |
| RSA | < 624 | SHA256withRSA (key size is too small for using SHA−384) |
| | <= 7680 | SHA384withRSA |
| | > 7680 | SHA512withRSA |
| EC | < 512 | SHA384withECDSA |
| | >= 512 | SHA512withECDSA |
| RSASSA−PSS | < 624 | RSASSA−PSS (with SHA−256, key size is too small for using SHA−384) |
| | <= 7680 | RSASSA−PSS (with SHA−384) |
| | > 7680 | RSASSA−PSS (with SHA−512) |
| EdDSA | 255 | Ed25519 |

| | | |
|---|---|---|
| | 448 | Ed448 |
| Ed25519 | 255 | Ed25519 |
| Ed448 | 448 | Ed448 |

- The key size, measured in bits, corresponds to the size of the private key. This size is determined by the value of the `-keysize` or `-groupname` options or the value derived from a default setting.
- An RSASSA-PSS signature algorithm uses a `MessageDigest` algorithm as its hash and MGF1 algorithms.
- If neither a default `-keysize` or `-groupname` is defined for an algorithm, the security provider will choose a default setting.

**Note:**

To improve out of the box security, default keysize, groupname, and signature algorithm names are periodically updated to stronger values with each release of the JDK. If interoperability with older releases of the JDK is important, make sure that the defaults are supported by those releases. Alternatively, you can use the `-keysize`, `-groupname`, or `-sigalg` options to override the default values at your own risk.

## SUPPORTED NAMED EXTENSIONS

The `keytool` command supports these named extensions. The names aren't case-sensitive.

`BC` or `BasicConstraints`

Values:

The full form is `ca:{true|false}[,pathlen:`*len*`]` or *len*, which is short for `ca:true,pathlen:`*len*.

When *len* is omitted, the resulting value is `ca:true`.

`KU` or `KeyUsage`

Values:

*usage*`(,` *usage*`)*`

*usage* can be one of the following:

- `digitalSignature`
- `nonRepudiation` (`contentCommitment`)
- `keyEncipherment`
- `dataEncipherment`
- `keyAgreement`
- `keyCertSign`
- `cRLSign`
- `encipherOnly`
- `decipherOnly`

Provided there is no ambiguity, the *usage* argument can be abbreviated with the first few letters (such as `dig` for `digitalSignature`) or in camel-case style (such as `dS` for `digitalSignature` or `cRLS` for `cRLSign`). The *usage* values are case-sensitive.

`EKU` or `ExtendedKeyUsage`

Values:

*usage*`(,` *usage*`)*`

*usage* can be one of the following:

- `anyExtendedKeyUsage`

- `serverAuth`
- `clientAuth`
- `codeSigning`
- `emailProtection`
- `timeStamping`
- `OCSPSigning`
- Any OID string

Provided there is no ambiguity, the *usage* argument can be abbreviated with the first few letters or in camel−case style. The *usage* values are case−sensitive.

`SAN` or `SubjectAlternativeName`
Values:

*type*`:`*value*(`,` *type*`:`*value*)*

*type* can be one of the following:

- `EMAIL`
- `URI`
- `DNS`
- `IP`
- `OID`

The *value* argument is the string format value for the *type*.

`IAN` or `IssuerAlternativeName`
Values:

Same as `SAN` or `SubjectAlternativeName`.

`SIA` or `SubjectInfoAccess`
Values:

*method*`:`*location−type*`:`*location−value*(`,` *method*`:`*location−type*`:`*location−value*)*

*method* can be one of the following:

- `timeStamping`
- `caRepository`
- Any OID

The *location−type* and *location−value* arguments can be any *type*`:`*value* supported by the `SubjectAlternativeName` extension.

`AIA` or `AuthorityInfoAccess`
Values:

Same as `SIA` or `SubjectInfoAccess`.

The *method* argument can be one of the following:

- `ocsp`
- `caIssuers`
- Any OID

When *name* is OID, the value is the hexadecimal dumped Definite Encoding Rules (DER) encoding of the `extnValue` for the extension excluding the OCTET STRING type and length bytes. Other than standard hexadecimal numbers (0−9, a−f, A−F), any extra characters are ignored in the HEX string. Therefore, both 01:02:03:04 and 01020304 are accepted as identical values. When there is no value, the extension has an

empty value field.

A special name `honored`, used only in `-gencert`, denotes how the extensions included in the certificate request should be honored. The value for this name is a comma−separated list of `all` (all requested extensions are honored), *name*`{:[critical|non-critical]}` (the named extension is honored, but it uses a different `isCritical` attribute), and `-name` (used with `all`, denotes an exception). Requested extensions aren't honored by default.

If, besides the`-ext honored` option, another named or OID `-ext` option is provided, this extension is added to those already honored. However, if this name (or OID) also appears in the honored value, then its value and criticality override that in the request. If an extension of the same type is provided multiple times through either a name or an OID, only the last extension is used.

The `subjectKeyIdentifier` extension is always created. For non−self−signed certificates, the `authorityKeyIdentifier` is created.

**CAUTION:**

Users should be aware that some combinations of extensions (and other certificate fields) may not conform to the Internet standard. See **Certificate Conformance Warning**.

## EXAMPLES OF TASKS IN CREATING A KEYSTORE

The following examples describe the sequence actions in creating a keystore for managing public/private key pairs and certificates from trusted entities.

- **Generating the Key Pair**
- **Requesting a Signed Certificate from a CA**
- **Importing a Certificate for the CA**
- **Importing the Certificate Reply from the CA**
- **Exporting a Certificate That Authenticates the Public Key**
- **Importing the Keystore**
- **Generating Certificates for an SSL Server**

## GENERATING THE KEY PAIR

Create a keystore and then generate the key pair.

You can enter the command as a single line such as the following:

```
keytool -genkeypair -dname "cn=myname, ou=mygroup, o=mycompany,
c=mycountry" -alias business -keyalg rsa -keypass password -keystore
/working/mykeystore -storepass password -validity 180
```

The command creates the keystore named `mykeystore` in the working directory (provided it doesn't already exist), and assigns it the password specified by `-keypass`. It generates a public/private key pair for the entity whose distinguished name is `myname`, `mygroup`, `mycompany`, and a two−letter country code of `mycountry`. It uses the RSA key generation algorithm to create the keys; both are 3072 bits.

The command uses the default SHA384withRSA signature algorithm to create a self−signed certificate that includes the public key and the distinguished name information. The certificate is valid for 180 days, and is associated with the private key in a keystore entry referred to by `-alias business`. The private key is assigned the password specified by `-keypass`.

The command is significantly shorter when the option defaults are accepted. In this case, only `-keyalg` is required, and the defaults are used for unspecified options that have default values. You are prompted for any required values. You could have the following:

```
keytool -genkeypair -keyalg rsa
```

In this case, a keystore entry with the alias `mykey` is created, with a newly generated key pair and a certificate that is valid for 90 days. This entry is placed in your home directory in a keystore named `.keystore` . `.keystore` is created if it doesn't already exist. You are prompted for the distinguished name

information, the keystore password, and the private key password.

**Note:**

The rest of the examples assume that you responded to the prompts with values equal to those specified in the first `-genkeypair` command. For example, a distinguished name of `cn=`*myname*`, ou=`*mygroup*`, o=`*mycompany*`, c=`*mycountry*.

## REQUESTING A SIGNED CERTIFICATE FROM A CA

**Note:**

Generating the key pair created a self−signed certificate; however, a certificate is more likely to be trusted by others when it is signed by a CA.

To get a CA signature, complete the following process:

1. Generate a CSR:

   ```
   keytool -certreq -file myname.csr
   ```

   This creates a CSR for the entity identified by the default alias `mykey` and puts the request in the file named `myname.csr`.

2. Submit `myname.csr` to a CA, such as DigiCert.

The CA authenticates you, the requestor (usually offline), and returns a certificate, signed by them, authenticating your public key. In some cases, the CA returns a chain of certificates, each one authenticating the public key of the signer of the previous certificate in the chain.

## IMPORTING A CERTIFICATE FOR THE CA

To import a certificate for the CA, complete the following process:

1. Before you import the certificate reply from a CA, you need one or more trusted certificates either in your keystore or in the `cacerts` keystore file. See `-importcert` in **Commands**.

   - If the certificate reply is a certificate chain, then you need the top certificate of the chain. The root CA certificate that authenticates the public key of the CA.
   - If the certificate reply is a single certificate, then you need a certificate for the issuing CA (the one that signed it). If that certificate isn't self−signed, then you need a certificate for its signer, and so on, up to a self−signed root CA certificate.

   The `cacerts` keystore ships with a set of root certificates issued by the CAs of **the Oracle Java Root Certificate program** [https://www.oracle.com/java/technologies/javase/carootcertsprogram.html]. If you request a signed certificate from a CA, and a certificate authenticating that CA's public key hasn't been added to `cacerts`, then you must import a certificate from that CA as a trusted certificate.

   A certificate from a CA is usually self−signed or signed by another CA. If it is signed by another CA, you need a certificate that authenticates that CA's public key.

   For example, you have obtained a *X*`.cer` file from a company that is a CA and the file is supposed to be a self−signed certificate that authenticates that CA's public key. Before you import it as a trusted certificate, you should ensure that the certificate is valid by:

   1. Viewing it with the `keytool -printcert` command or the `keytool -importcert` command without using the `-noprompt` option. Make sure that the displayed certificate fingerprints match the expected fingerprints.
   2. Calling the person who sent the certificate, and comparing the fingerprints that you see with the ones that they show or that a secure public key repository shows.

   Only when the fingerprints are equal is it assured that the certificate wasn't replaced in transit with somebody else's certificate (such as an attacker's certificate). If such an attack takes place, and you didn't check the certificate before you imported it, then you would be trusting anything that the attacker signed.

2. Replace the self−signed certificate with a certificate chain, where each certificate in the chain authenticates the public key of the signer of the previous certificate in the chain, up to a root CA.

   If you trust that the certificate is valid, then you can add it to your keystore by entering the following command:

   ```
   keytool −importcert −alias alias −file *X*.cer`
   ```

   This command creates a trusted certificate entry in the keystore from the data in the CA certificate file and assigns the values of the *alias* to the entry.

## IMPORTING THE CERTIFICATE REPLY FROM THE CA

After you import a certificate that authenticates the public key of the CA that you submitted your certificate signing request to (or there is already such a certificate in the `cacerts` file), you can import the certificate reply and replace your self−signed certificate with a certificate chain.

The certificate chain is one of the following:

- Returned by the CA when the CA reply is a chain.
- Constructed when the CA reply is a single certificate. This certificate chain is constructed by using the certificate reply and trusted certificates available either in the keystore where you import the reply or in the `cacerts` keystore file.

For example, if you sent your certificate signing request to DigiCert, then you can import their reply by entering the following command:

**Note:**

In this example, the returned certificate is named `DCmyname.cer`.

```
keytool −importcert −trustcacerts −file DCmyname.cer
```

## EXPORTING A CERTIFICATE THAT AUTHENTICATES THE PUBLIC KEY

**Note:**

If you used the `jarsigner` command to sign a Java Archive (JAR) file, then clients that use the file will want to authenticate your signature.

One way that clients can authenticate you is by importing your public key certificate into their keystore as a trusted entry. You can then export the certificate and supply it to your clients.

For example:

1. Copy your certificate to a file named `myname.cer` by entering the following command:

   **Note:**

   In this example, the entry has an alias of `mykey`.

   ```
   keytool −exportcert −alias mykey −file myname.cer
   ```

2. With the certificate and the signed JAR file, a client can use the `jarsigner` command to authenticate your signature.

## IMPORTING THE KEYSTORE

Use the `importkeystore` command to import an entire keystore into another keystore. This imports all entries from the source keystore, including keys and certificates, to the destination keystore with a single command. You can use this command to import entries from a different type of keystore. During the import, all new entries in the destination keystore will have the same alias names and protection passwords (for secret keys and private keys). If the `keytool` command can't recover the private keys or secret keys from the source keystore, then it prompts you for a password. If it detects alias duplication, then it asks you for a new alias, and you can specify a new alias or simply allow the `keytool` command to overwrite the existing one.

For example, import entries from a typical JKS type keystore `key.jks` into a PKCS #11 type hardware−based keystore, by entering the following command:

```
keytool -importkeystore -srckeystore key.jks -destkeystore NONE
-srcstoretype JKS -deststoretype PKCS11 -srcstorepass password
-deststorepass password
```

The `importkeystore` command can also be used to import a single entry from a source keystore to a destination keystore. In this case, besides the options you used in the previous example, you need to specify the alias you want to import. With the `-srcalias` option specified, you can also specify the destination alias name, protection password for a secret or private key, and the destination protection password you want as follows:

```
keytool -importkeystore -srckeystore key.jks -destkeystore NONE
-srcstoretype JKS -deststoretype PKCS11 -srcstorepass password
-deststorepass password -srcalias myprivatekey -destalias myoldpri-
vatekey -srckeypass password -destkeypass password -noprompt
```

## GENERATING CERTIFICATES FOR AN SSL SERVER

The following are `keytool` commands used to generate key pairs and certificates for three entities:

- Root CA (`root`)
- Intermediate CA (`ca`)
- SSL server (`server`)

Ensure that you store all the certificates in the same keystore.

```
keytool -genkeypair -keystore root.jks -alias root -ext bc:c -keyalg rsa
keytool -genkeypair -keystore ca.jks -alias ca -ext bc:c -keyalg rsa
keytool -genkeypair -keystore server.jks -alias server -keyalg rsa

keytool -keystore root.jks -alias root -exportcert -rfc > root.pem

keytool -storepass password -keystore ca.jks -certreq -alias ca |
    keytool -storepass password -keystore root.jks
    -gencert -alias root -ext BC=0 -rfc > ca.pem
keytool -keystore ca.jks -importcert -alias ca -file ca.pem

keytool -storepass password -keystore server.jks -certreq -alias server |
    keytool -storepass password -keystore ca.jks -gencert -alias ca
    -ext ku:c=dig,kE -rfc > server.pem
cat root.pem ca.pem server.pem |
    keytool -keystore server.jks -importcert -alias server
```

## TERMS

Keystore

A keystore is a storage facility for cryptographic keys and certificates.

Keystore entries

Keystores can have different types of entries. The two most applicable entry types for the `keytool` command include the following:

Key entries: Each 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 for the corresponding public key. See **Certificate Chains**. The `keytool` command can handle both types of entries, while the `jarsigner` tool only handles the latter type of entry, that is private keys and their associated certificate chains.

Trusted certificate entries: Each entry contains a single public key certificate that belongs to another party. The entry is called a trusted certificate because the keystore owner trusts that the public key in the certificate belongs to the identity identified by the subject (owner) of the certificate. The issuer of the certificate vouches for this, by signing the certificate.

Keystore aliases

All keystore entries (key and trusted certificate entries) are accessed by way of unique aliases.

An alias is specified when you add an entity to the keystore with the `-genseckey` command to generate a secret key, the `-genkeypair` command to generate a key pair (public and private key), or the `-importcert` command to add a certificate or certificate chain to the list of trusted certificates. Subsequent `keytool` commands must use this same alias to refer to the entity.

For example, you can use the alias `duke` to generate a new public/private key pair and wrap the public key into a self−signed certificate with the following command. See **Certificate Chains**.

```
keytool -genkeypair -alias duke -keyalg rsa -keypass passwd
```

This example specifies an initial *passwd* required by subsequent commands to access the private key associated with the alias `duke`. If you later want to change Duke's private key password, use a command such as the following:

```
keytool -keypasswd -alias duke -keypass passwd -new newpasswd
```

This changes the initial *passwd* to *newpasswd*. A password shouldn't be specified on a command line or in a script unless it is for testing purposes, or you are on a secure system. If you don't specify a required password option on a command line, then you are prompted for it.

Keystore implementation

The `KeyStore` class provided in the `java.security` package 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, two command−line tools (`keytool` and `jarsigner`) make use of keystore implementations. Because the `KeyStore` class is `public`, users can write additional security applications that use it.

In JDK 9 and later, the default 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. There is another built−in implementation, provided by Oracle. It implements the keystore as a file with 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 Service Provider Interface 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 Security API. To provide a keystore implementation, clients must implement a provider and supply a `KeystoreSpi` subclass implementation, as described in Steps to Implement and Integrate a Provider.

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

The `keytool` command works on any file−based keystore implementation. It treats the keystore location that is passed to it at the command line as a file name and converts it to a `FileInputStream`, from which it loads the keystore information. The `jarsigner` commands can read a keystore from any location that can be specified with a URL.

For `keytool` and `jarsigner`, you can specify a keystore type at the command line, with the

`-storetype` option.

If you don't explicitly specify a keystore type, then the tools choose a keystore implementation based on the value of the `keystore.type` property specified in the security properties file. The security properties file is called `java.security`, and resides in the security properties directory:

- **Linux and macOS:** `java.home/lib/security`
- **Windows:** `java.home\lib\security`

Each tool gets the `keystore.type` value and then examines all the currently installed providers until it finds one that implements a keystores of that type. It then uses the keystore implementation from that provider.The `KeyStore` class defines a static method named `getDefaultType` that lets applications retrieve the value of the `keystore.type` property. The following line of code creates an instance of the default keystore type as specified in the `keystore.type` property:

```
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDe-
faultType());
```

The default keystore type is `pkcs12`, which is a cross−platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This is specified by the following line in the security properties file:

```
keystore.type=pkcs12
```

To have the tools utilize a keystore implementation other than the default, you can change that line to specify a different keystore type. For example, if you want to use the Oracle's `jks` keystore implementation, then change the line to the following:

```
keystore.type=jks
```

**Note:**

Case doesn't matter in keystore type designations. For example, `JKS` would be considered the same as `jks`.

Certificate

A certificate (or public−key certificate) is a digitally signed statement from one entity (the issuer), saying that the public key and some other information of another entity (the subject) has some specific value. The following terms are related to certificates:

- Public Keys: These are numbers associated with a particular entity, and are intended to be known to everyone who needs to have trusted interactions with that entity. Public keys are used to verify signatures.
- Digitally Signed: If some data is digitally signed, then it is stored with the identity of an entity and a signature that proves that entity knows about the data. The data is rendered unforgeable by signing with the entity's private key.
- Identity: A known way of addressing an entity. In some systems, the identity is the public key, and in others it can be anything from an Oracle Solaris UID to an email address to an X.509 distinguished name.
- Signature: A signature is computed over some data using the private key of an entity. The signer, which in the case of a certificate is also known as the issuer.
- Private Keys: These are numbers, each of which is supposed to be known only to the particular entity whose private key it is (that is, it is supposed to be kept secret). Private and public keys exist in pairs in all public key cryptography systems (also referred to as public key crypto systems). In a typical public key crypto system, such as DSA, a private key corresponds to exactly one public key. Private keys are used to compute signatures.

- Entity: An entity is a person, organization, program, computer, business, bank, or something else you are trusting to some degree.

Public key cryptography requires access to users' public keys. In a large−scale networked environment, it is impossible to guarantee that prior relationships between communicating entities were established or that a trusted repository exists with all used public keys. Certificates were invented as a solution to this public key distribution problem. Now a Certification Authority (CA) can act as a trusted third party. CAs are entities such as businesses that are trusted to sign (issue) certificates for other entities. It is assumed that CAs only create valid and reliable certificates because they are bound by legal agreements. There are many public Certification Authorities, such as DigiCert, Comodo, Entrust, and so on.

You can also run your own Certification Authority using products such as Microsoft Certificate Server or the Entrust CA product for your organization. With the `keytool` command, it is possible to display, import, and export certificates. It is also possible to generate self−signed certificates.

The `keytool` command currently handles X.509 certificates.

X.509 Certificates

The X.509 standard defines what information can go into a certificate and describes how to write it down (the data format). All the data in a certificate is encoded with two related standards called ASN.1/DER. Abstract Syntax Notation 1 describes data. The Definite Encoding Rules describe a single way to store and transfer that data.

All X.509 certificates have the following data, in addition to the signature:

- Version: This identifies which version of the X.509 standard applies to this certificate, which affects what information can be specified in it. Thus far, three versions are defined. The `keytool` command can import and export v1, v2, and v3 certificates. It generates v3 certificates.
  - X.509 Version 1 has been available since 1988, is widely deployed, and is the most generic.
  - X.509 Version 2 introduced the concept of subject and issuer unique identifiers to handle the possibility of reuse of subject or issuer names over time. Most certificate profile documents strongly recommend that names not be reused and that certificates shouldn't make use of unique identifiers. Version 2 certificates aren't widely used.
  - X.509 Version 3 is the most recent (1996) and supports the notion of extensions where anyone can define an extension and include it in the certificate. Some common extensions are: KeyUsage (limits the use of the keys to particular purposes such as `signing-only`) and AlternativeNames (allows other identities to also be associated with this public key, for example. DNS names, email addresses, IP addresses). Extensions can be marked critical to indicate that the extension should be checked and enforced or used. For example, if a certificate has the KeyUsage extension marked critical and set to `keyCertSign`, then when this certificate is presented during SSL communication, it should be rejected because the certificate extension indicates that the associated private key should only be used for signing certificates and not for SSL use.
- Serial number: The entity that created the certificate is responsible for assigning it a serial number to distinguish it from other certificates it issues. This information is used in numerous ways. For example, when a certificate is revoked its serial number is placed in a Certificate Revocation List (CRL).
- Signature algorithm identifier: This identifies the algorithm used by the CA to sign the certificate.
- Issuer name: The X.500 Distinguished Name of the entity that signed the certificate. This is typically a CA. Using this certificate implies trusting the entity that signed this certificate. In some cases, such as root or top−level CA certificates, the issuer signs its own certificate.

- Validity period: Each certificate is valid only for a limited amount of time. This period is described by a start date and time and an end date and time, and can be as short as a few seconds or almost as long as a century. The validity period chosen depends on a number of factors, such as the strength of the private key used to sign the certificate, or the amount one is willing to pay for a certificate. This is the expected period that entities can rely on the public value, when the associated private key has not been compromised.
- Subject name: The name of the entity whose public key the certificate identifies. This name uses the X.500 standard, so it is intended to be unique across the Internet. This is the X.500 Distinguished Name (DN) of the entity. For example,

  ```
  CN=Java Duke, OU=Java Software Division, O=Oracle Corporation, C=US
  ```

  These refer to the subject's common name (CN), organizational unit (OU), organization (O), and country (C).
- Subject public key information: This is the public key of the entity being named with an algorithm identifier that specifies which public key crypto system this key belongs to and any associated key parameters.

Certificate Chains

The `keytool` command can create and manage keystore key entries that each contain a private key and an associated certificate chain. The first certificate in the chain contains the public key that corresponds to the private key.

When keys are first generated, the chain usually starts off containing a single element, a self−signed certificate. See −genkeypair in **Commands**. A self−signed certificate is one for which the issuer (signer) is the same as the subject. The subject is the entity whose public key is being authenticated by the certificate. When the `-genkeypair` command is called to generate a new public/private key pair, it also wraps the public key into a self−signed certificate (unless the `-signer` option is specified).

Later, after a Certificate Signing Request (CSR) was generated with the `-certreq` command and sent to a Certification Authority (CA), the response from the CA is imported with `-importcert`, and the self−signed certificate is replaced by a chain of certificates. At the bottom of the chain is the certificate (reply) issued by the CA authenticating the subject's public key. The next certificate in the chain is one that authenticates the CA's public key.

In many cases, this is a self−signed certificate, which is a certificate from the CA authenticating its own public key, and the last certificate in the chain. In other cases, the CA might return a chain of certificates. In this case, the bottom certificate in the chain is the same (a certificate signed by the CA, authenticating the public key of the key entry), but the second certificate in the chain is a certificate signed by a different CA that authenticates the public key of the CA you sent the CSR to. The next certificate in the chain is a certificate that authenticates the second CA's key, and so on, until a self−signed root certificate is reached. Each certificate in the chain (after the first) authenticates the public key of the signer of the previous certificate in the chain.

Many CAs only return the issued certificate, with no supporting chain, especially when there is a flat hierarchy (no intermediates CAs). In this case, the certificate chain must be established from trusted certificate information already stored in the keystore.

A different reply format (defined by the PKCS #7 standard) includes the supporting certificate chain in addition to the issued certificate. Both reply formats can be handled by the `keytool` command.

The top−level (root) CA certificate is self−signed. However, the trust into the root's public key doesn't come from the root certificate itself, but from other sources such as a newspaper. This is because anybody could generate a self−signed certificate with the distinguished name of, for example, the DigiCert root CA. The root CA public key is widely known. The only reason it is stored in a certificate is because this is the format understood by most tools, so the certificate in

this case is only used as a vehicle to transport the root CA's public key. Before you add the root CA certificate to your keystore, you should view it with the `-printcert` option and compare the displayed fingerprint with the well-known fingerprint obtained from a newspaper, the root CA's Web page, and so on.

cacerts Certificates File

A certificates file named `cacerts` resides in the security properties directory:

- **Linux and macOS:** *JAVA_HOME*`/lib/security`
- **Windows:** *JAVA_HOME*`\lib\security`

The `cacerts` file represents a system-wide keystore with CA certificates. System administrators can configure and manage that file with the `keytool` command by specifying `jks` as the keystore type. The `cacerts` keystore file ships with a default set of root CA certificates. For Linux, macOS, and Windows, you can list the default certificates with the following command:

```
keytool -list -cacerts
```

The initial password of the `cacerts` keystore file is `changeit`. System administrators should change that password and the default access permission of that file upon installing the SDK.

**Note:**

It is important to verify your `cacerts` file. Because you trust the CAs in the `cacerts` file as entities for signing and issuing certificates to other entities, you must manage the `cacerts` file carefully. The `cacerts` file should contain only certificates of the CAs you trust. It is your responsibility to verify the trusted root CA certificates bundled in the `cacerts` file and make your own trust decisions.

To remove an untrusted CA certificate from the `cacerts` file, use the `-delete` option of the `keytool` command. You can find the `cacerts` file in the JDK's `$JAVA_HOME/lib/security` directory. Contact your system administrator if you don't have permission to edit this file.

Internet RFC 1421 Certificate Encoding Standard

Certificates are often stored using the printable encoding format defined by the Internet RFC 1421 standard, instead of their binary encoding. This certificate format, also known as Base64 encoding, makes it easy to export certificates to other applications by email or through some other mechanism.

Certificates read by the `-importcert` and `-printcert` commands can be in either this format or binary encoded. The `-exportcert` command by default outputs a certificate in binary encoding, but will instead output a certificate in the printable encoding format, when the `-rfc` option is specified.

The `-list` command by default prints the SHA-256 fingerprint of a certificate. If the `-v` option is specified, then the certificate is printed in human-readable format. If the `-rfc` option is specified, then the certificate is output in the printable encoding format.

In its printable encoding format, the encoded certificate is bounded at the beginning and end by the following text:

```
-----BEGIN CERTIFICATE-----

encoded certificate goes here.

-----END CERTIFICATE-----
```

X.500 Distinguished Names

X.500 Distinguished Names are used to identify entities, such as those that are named by the `subject` and `issuer` (signer) fields of X.509 certificates. The `keytool` command supports the following subparts:

- commonName: The common name of a person such as Susan Jones.
- organizationUnit: The small organization (such as department or division) name. For example, Purchasing.
- localityName: The locality (city) name, for example, Palo Alto.
- stateName: State or province name, for example, California.
- country: Two−letter country code, for example, CH.

When you supply a distinguished name string as the value of a `-dname` option, such as for the `-genkeypair` command, the string must be in the following format:

```
CN=cName, OU=orgUnit, O=org, L=city, S=state, C=countryCode
```

All the following items represent actual values and the previous keywords are abbreviations for the following:

```
CN=commonName
OU=organizationUnit
O=organizationName
L=localityName
S=stateName
C=country
```

A sample distinguished name string is:

```
CN=Mark Smith, OU=Java, O=Oracle, L=Cupertino, S=California,
C=US
```

A sample command using such a string is:

```
keytool -genkeypair -dname "CN=Mark Smith, OU=Java, O=Ora-
cle, L=Cupertino, S=California, C=US" -alias mark -keyalg
rsa
```

Case doesn't matter for the keyword abbreviations. For example, CN, cn, and Cn are all treated the same.

Order matters; each subcomponent must appear in the designated order. However, it isn't necessary to have all the subcomponents. You can use a subset, for example:

```
CN=Smith, OU=Java, O=Oracle, C=US
```

If a distinguished name string value contains a comma, then the comma must be escaped by a backslash (\) character when you specify the string on a command line, as in:

```
cn=Jack, ou=Java\, Product Development, o=Oracle, c=US
```

It is never necessary to specify a distinguished name string on a command line. When the distinguished name is needed for a command, but not supplied on the command line, the user is prompted for each of the subcomponents. In this case, a comma doesn't need to be escaped by a backslash (\).

## WARNINGS

### IMPORTING TRUSTED CERTIFICATES WARNING

**Important**: Be sure to check a certificate very carefully before importing it as a trusted certificate.

**Windows Example:**

View the certificate first with the `-printcert` command or the `-importcert` command without the `-noprompt` option. Ensure that the displayed certificate fingerprints match the expected ones. For example, suppose someone sends or emails you a certificate that you put it in a file named `\tmp\cert`. Before you consider adding the certificate to your list of trusted certificates, you can execute a `-printcert` command to view its fingerprints, as follows:

```
keytool -printcert -file \tmp\cert
  Owner: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
  Issuer: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
  Serial Number: 59092b34
  Valid from: Thu Jun 24 18:01:13 PDT 2016 until: Wed Jun 23 17:01:13 PST
  Certificate Fingerprints:

               SHA-1: 20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:1
             SHA-256: 90:7B:70:0A:EA:DC:16:79:92:99:41:FF:8A:FE:EB:90:
                      17:75:E0:90:B2:24:4D:3A:2A:16:A6:E4:11:0F:67:A4
```

**Linux Example:**

View the certificate first with the `-printcert` command or the `-importcert` command without the `-noprompt` option. Ensure that the displayed certificate fingerprints match the expected ones. For example, suppose someone sends or emails you a certificate that you put it in a file named `/tmp/cert`. Before you consider adding the certificate to your list of trusted certificates, you can execute a `-printcert` command to view its fingerprints, as follows:

```
keytool -printcert -file /tmp/cert
  Owner: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
  Issuer: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
  Serial Number: 59092b34
  Valid from: Thu Jun 24 18:01:13 PDT 2016 until: Wed Jun 23 17:01:13 PST
  Certificate Fingerprints:

               SHA-1: 20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:1
               SHA-256: 90:7B:70:0A:EA:DC:16:79:92:99:41:FF:8A:FE:EB:90
                        17:75:E0:90:B2:24:4D:3A:2A:16:A6:E4:11:0F:67:A4
```

Then call or otherwise contact the person who sent the certificate and compare the fingerprints that you see with the ones that they show. Only when the fingerprints are equal is it guaranteed that the certificate wasn't replaced in transit with somebody else's certificate such as an attacker's certificate. If such an attack took place, and you didn't check the certificate before you imported it, then you would be trusting anything the attacker signed, for example, a JAR file with malicious class files inside.

**Note:**

It isn't required that you execute a `-printcert` command before importing a certificate. This is because before you add a certificate to the list of trusted certificates in the keystore, the `-importcert` command prints out the certificate information and prompts you to verify it. You can then stop the import operation. However, you can do this only when you call the `-importcert` command without the `-noprompt` option. If the `-noprompt` option is specified, then there is no interaction with the user.

## PASSWORDS WARNING

Most commands that operate on a keystore require the store password. Some commands require a private/secret key password. Passwords can be specified on the command line in the `-storepass` and `-keypass` options. However, a password shouldn't be specified on a command line or in a script unless it is for testing, or you are on a secure system. When you don't specify a required password option on a command line, you are prompted for it.

## CERTIFICATE CONFORMANCE WARNING

**Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile** [https://tools.ietf.org/rfc/rfc5280.txt] defined a profile on conforming X.509 certificates, which includes what values and value combinations are valid for certificate fields and extensions.

The `keytool` command doesn't enforce all of these rules so it can generate certificates that don't conform to the standard, such as self–signed certificates that would be used for internal testing purposes. Certificates that don't conform to the standard might be rejected by the JDK or other applications. Users should ensure that they provide the correct options for `-dname`, `-ext`, and so on.

## IMPORT A NEW TRUSTED CERTIFICATE

Before you add the certificate to the keystore, the `keytool` command verifies it by attempting to construct a chain of trust from that certificate to a self−signed certificate (belonging to a root CA), using trusted certificates that are already available in the keystore.

If the `−trustcacerts` option was specified, then additional certificates are considered for the chain of trust, namely the certificates in a file named `cacerts`.

If the `keytool` command fails to establish a trust path from the certificate to be imported up to a self−signed certificate (either from the keystore or the `cacerts` file), then the certificate information is printed, and the user is prompted to verify it by comparing the displayed certificate fingerprints with the fingerprints obtained from some other (trusted) source of information, which might be the certificate owner. Be very careful to ensure the certificate is valid before importing it as a trusted certificate. The user then has the option of stopping the import operation. If the `−noprompt` option is specified, then there is no interaction with the user.

## IMPORT A CERTIFICATE REPLY

When you import a certificate reply, the certificate reply is validated with trusted certificates from the keystore, and optionally, the certificates configured in the `cacerts` keystore file when the `−trustcacerts` option is specified.

The methods of determining whether the certificate reply is trusted are as follows:

- If the reply is a single X.509 certificate, then the `keytool` command attempts to establish a trust chain, starting at the certificate reply and ending at a self−signed certificate (belonging to a root CA). The certificate reply and the hierarchy of certificates is used to authenticate the certificate reply from the new certificate chain of aliases. If a trust chain can't be established, then the certificate reply isn't imported. In this case, the `keytool` command doesn't print the certificate and prompt the user to verify it, because it is very difficult for a user to determine the authenticity of the certificate reply.
- If the reply is a PKCS #7 formatted certificate chain or a sequence of X.509 certificates, then the chain is ordered with the user certificate first followed by zero or more CA certificates. If the chain ends with a self−signed root CA certificate and the`−trustcacerts` option was specified, the `keytool` command attempts to match it with any of the trusted certificates in the keystore or the `cacerts` keystore file. If the chain doesn't end with a self−signed root CA certificate and the `−trustcacerts` option was specified, the `keytool` command tries to find one from the trusted certificates in the keystore or the `cacerts` keystore file and add it to the end of the chain. If the certificate isn't found and the `−noprompt` option isn't specified, the information of the last certificate in the chain is printed, and the user is prompted to verify it.

If the public key in the certificate reply matches the user's public key already stored with `alias`, then the old certificate chain is replaced with the new certificate chain in the reply. The old chain can only be replaced with a valid `keypass`, and so the password used to protect the private key of the entry is supplied. If no password is provided, and the private key password is different from the keystore password, the user is prompted for it.

This command was named `−import` in earlier releases. This old name is still supported in this release. The new name, `−importcert`, is preferred.