keytool-Key and Certificate Management Tool

The keytool command interface has changed in Java SE 6. See the Changes Section for a detailed description. Note that previously defined commands are still supported.

DESCRIPTION

keytool is a key and certificate management utility. It allows users to administer their own public/private key pairs and associated certificates for use in self-authentication (where the user authenticates himself/herself to other users/services) or data integrity and authentication services, using digital signatures. It also allows 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, etc.), saying that the public key (and some other information) of some other entity has a particular value. (See Certificates.) When data is digitally signed, the signature can be verified to check the data integrity and authenticity. *Integrity* means that the data has not been modified or tampered with, and *authenticity* means the data indeed comes from whoever claims to have created and signed it.

keytool also enables users to administer secret keys used in symmetric encryption/decryption (e.g. DES).

keytool stores the keys and certificates in a *keystore*.

COMMAND AND OPTION NOTES

The various commands and their options are listed and described below. Note:

- All command and option names are preceded by a minus sign (-).
- The options for each command may be provided in any order.
- All items not italicized or in braces or square brackets are required to appear as is.
- Braces surrounding an option generally signify that a default value will be used if the option is not specified on the command line. Braces are also used around the `-v`, `-`

`rfc`, and `-J` options, which only have meaning if they appear on the command line (that is, they don't have any "default" values other than not existing).
- Brackets surrounding an option signify that the user is prompted for the value(s) if the option is not specified on the command line. (For a `-keypass` option, if you do not specify the option on the command line, **keytool** will first attempt to use the keystore password to recover the private/secret key, and if this fails, will then prompt 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 specifying a `-printcert` command, replace *cert_file* with the actual file name, as in:

  ```
  keytool -printcert -file VScert.cer
  ```

- Option values must be quoted if they contain a blank (space).
- The `-help` command is the default. Thus, the command line

  ```
  keytool
  ```

  is equivalent to

  ```
  keytool -help
  ```

Option Defaults

Below are the defaults for various option values.

```
-alias "mykey"

-keyalg
    "DSA" (when using -genkeypair)
    "DES" (when using -genseckey)

-keysize
    2048 (when using -genkeypair and -keyalg is "RSA")
    1024 (when using -genkeypair and -keyalg is "DSA")
```

```
    256 (when using -genkeypair and -keyalg is "EC")
    56 (when using -genseckey and -keyalg is "DES")
    168 (when using -genseckey and -keyalg is "DESede")


-validity 90

-keystore 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
```

In generating a public/private key pair, the signature algorithm (-*sigalg* option) is derived from the algorithm of the underlying private key:

- If the underlying private key is of type "DSA", the -*sigalg* option defaults to "SHA1withDSA"
- If the underlying private key is of type "RSA", the -*sigalg* option defaults to "SHA256withRSA".
- If the underlying private key is of type "EC", the -*sigalg* option defaults to "SHA256withECDSA".

Please consult the Java Cryptography Architecture API Specification & Reference for a full list of -*keyalg* and -*sigalg* you can choose from.

Common Options

The `-v` option can appear for all commands except `-help`. If it appears, it signifies "verbose" mode; more information will be provided in the output.

There is also a `-J`*`javaoption`* option that may appear for any command. If it appears, the specified *javaoption* string is passed through directly to the Java interpreter. This option should not contain any spaces. It is useful for adjusting the execution environment or memory usage. For a list of possible interpreter options, type `java -h` or `java -X` at the command line.

These options may 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 does not yet exist, then certain **keytool** commands may result in a new keystore file being created. For example, if `keytool -genkeypair` is invoked and the `-keystore` option is not specified, the default keystore file named `.keystore` in the user's home directory will be created if it does not already exist. Similarly, if the `-keystore` *`ks_file`* option is specified but *ks_file* does not exist, then it will be created

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` is not file-based (for example, if it resides on a hardware token device).

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

If the modifier `env` or `file` is not specified, then the password has the value *argument*, which must be at least 6 characters long. 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, if a `-storepass` option is not provided at the command line, the user is prompted for it.

When retrieving information from the keystore, the password is optional; if no password is given, the integrity of the retrieved information cannot be checked and a warning is displayed.

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

`-providerClass` *`provider_class_name`*
Used to specify the name of cryptographic service provider's master class file when the service provider is not listed in the security properties file.

`-providerArg` *`provider_arg`*
Used in conjunction with `-providerClass`. Represents an optional string input argument for the constructor of *provider_class_name*.

`-protected`
Either `true` or `false`. This value should be specified as `true` if a password must be given via a protected authentication path such as a dedicated PIN reader.

Note: Since there are two keystores involved in `-importkeystore` command, two options, namely, `-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 certificate generated, or in `-certreq` to show what extensions are requested in the certificate request. The option can appear multiple times. name can be a supported extension name (see below) or an arbitrary OID number. value, if provided, denotes the parameter for the extension; if omitted, denotes the default value (if defined) of the extension or the extension requires no parameter. The `:critical` modifier, if provided, means the extension's isCritical attribute is true; otherwise, false. You may use `:c` in place of `:critical`.

Currently keytool supports these named extensions (case-insensitive):

Name	Value
BC or BasicConstraints	The full form: "ca:{true|false}[,pathlen:<len>]"; or, <len>, a shorthand for "ca:true,pathlen:<len>"; or omitted, means "ca:true"
KU or KeyUsage	usage(,usage)*, usage can be one of digitalSignature, nonRepudiation (contentCommitment), keyEncipherment, dataEncipherment, keyAgreement, keyCertSign, cRLSign, encipherOnly, decipherOnly. Usage can be abbreviated with the first few letters (say, dig for digitalSignature) or in camel-case style (say, dS for digitalSignature, cRLS for cRLSign), as long as no ambiguity is found. Usage is case-insensitive.
EKU or ExtendedkeyUsage	usage(,usage)*, usage can be one of anyExtendedKeyUsage, serverAuth, clientAuth, codeSigning, emailProtection, timeStamping, OCSPSigning, or any OID string. Named usage can be abbreviated with the first few letters or in camel-case style, as long as no ambiguity is found. Usage is

	case-insensitive.
SAN or SubjectAlternativeName	type:value(,type:value)*, type can be EMAIL, URI, DNS, IP, or OID, value is the string format value for the type.
IAN or IssuerAlternativeName	same as SubjectAlternativeName
SIA or SubjectInfoAccess	method:location-type:location-value (,method:location-type:location-value)*, method can be "timeStamping", "caRepository" or any OID. location-type and location-value can be any type:value supported by the SubjectAlternativeName extension.
AIA or AuthorityInfoAccess	same as SubjectInfoAccess. method can be "ocsp","caIssuers" or any OID.

For name as OID, value is the HEX dumped DER encoding of the extnValue for the extension excluding the OCTET STRING type and length bytes. Any extra character other than standard HEX numbers (0-9, a-f, A-F) are ignored in the HEX string. Therefore, both `"01:02:03:04"` and `"01020304"` are accepted as identical values. If there is no value, the extension has an empty value field then.

A special name `'honored'`, used in `-gencert` only, 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 using a different isCritical attribute) and `"-name"` (used with all, denotes an exception). Requested extensions are not honored by default.

If, besides the -ext honored option, another named or OID -ext option is provided, this extension will be added to those already honored. However, if this name (or OID) also appears in the honored value, its value and criticality overrides the one in the request.

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

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

COMMANDS

Creating or Adding Data to the Keystore

`-gencert {-rfc} {-infile infile} {-outfile outfile} {-alias alias} {-sigalg sigalg} {-dname dname} {-startdate startdate {-ext ext}* {-validity valDays} [-keypass keypass] {-keystore keystore} [-storepass storepass] {-storetype storetype} {-providername provider_name} {-providerClass provider_class_name {-providerArg provider_arg}} {-v} {-protected} {-Jjavaoption}`

Generates a certificate as a response to a certificate request file (which can be created by the `keytool -certreq` command). The command reads the request from *infile* (if omitted, from the standard input), signs it using alias's private key, and output the X.509 certificate into *outfile* (if omitted, to the standard output). If `-rfc` is specified, output format is BASE64-encoded PEM; otherwise, a binary DER is created.

sigalg specifies the algorithm that should be used to sign the certificate. *startdate* is the start time/date that the certificate is valid. *valDays* tells the number of days for which the certificate should be considered valid.

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

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

The `-gencert` command 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
keytool -alias ca1 -dname CN=CA -genkeypair
keytool -alias ca2 -dname CN=CA -genkeypair
keytool -alias e1 -dname CN=E1 -genkeypair
```

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 | $KT -alias ca1 -gencert -ext
san=dns:ca2 | $KT -alias ca2 -importcert
```

The following command creates the certificate `e1` and stores it in the file `e1.cert`, 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 {-alias *alias*} {-keyalg *keyalg*} {-keysize *keysize*} {-sigalg *sigalg*} [-dname *dname*] [-keypass *keypass*] {-startdate *value*} {-ext *ext*}* {-validity *valDays*} {-storetype *storetype*} {-keystore *keystore*} [-storepass *storepass*] {-providerClass *provider_class_name* {-providerArg *provider_arg*}} {-v} {-protected} {-J*javaoption*}

Generates a key pair (a public key and associated private key). Wraps the public key into an X.509 v3 self-signed certificate, which is stored as a single-element certificate chain. This certificate chain and the private key are stored in a new keystore entry identified by *alias*.

keyalg specifies the algorithm to be used to generate the key pair, and *keysize* specifies the size of each key to be generated. *sigalg* specifies the algorithm that should be used to sign the self-signed certificate; this algorithm must be compatible with *keyalg*.

dname specifies the X.500 Distinguished Name to be associated with *alias*, and is used as the `issuer` and `subject` fields in the self-signed certificate. If no distinguished name is provided at the command line, the user will be prompted for one.

keypass is a password used to protect the private key of the generated key pair. If no password is provided, the user is prompted for it. If you press RETURN at the prompt, the key password is set to the same password as that used for the keystore. *keypass* must be at least 6 characters long.

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:

1. ([+-]*nnn*[ymdHMS])+
2. [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 sub values. Inside each sub value, the plus sign ("+") means shifting forward, and the minus sign ("-") means shifting 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 using the `java.util.GregorianCalendar.add(int field, int amount)` method on each sub value, from left to right. For example, by specifying `"-startdate -1y+1m-1d"`, the issue time will be:

```
    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 may provide only one part, which means the other part is the same as the current date (or time). User must provide the exact number of digits as shown in the format definition (padding with 0 if shorter). When both the 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 is not provided, the start date is the current time. The option can be provided at most once.

valDays specifies the number of days (starting at the date specified by `-startdate`, or the current date if `-startdate` is not specified) for which the certificate should be considered valid.

This command was named `-genkey` in previous releases. This old name is still supported in this release and will be supported in future releases, but for clarity the new name, `-genkeypair`, is preferred going forward.

`-genseckey` `{-alias` *`alias`*`} {-keyalg` *`keyalg`*`} {-keysize` *`keysize`*`} [-keypass` *`keypass`*`] {-storetype` *`storetype`*`} {-keystore` *`keystore`*`} [-storepass` *`storepass`*`] {-providerClass` *`provider_class_name`* `{-providerArg` *`provider_arg`*`}} {-v} {-protected} {-J`*`javaoption`*`}`
Generates a secret key and stores it in a new `KeyStore.SecretKeyEntry` identified by *alias*.

keyalg specifies the algorithm to be used to generate the secret key, and *keysize* specifies the size of the key to be generated. *keypass* is a password used to protect the secret key. If no password is provided, the user is prompted for it. If you press RETURN at the prompt, the key password is set to the same password as that used for the keystore. *keypass* must be at least 6 characters long.

`-importcert` `{-alias` *`alias`*`} {-file` *`cert_file`*`} [-keypass` *`keypass`*`] {-noprompt} {-trustcacerts} {-storetype` *`storetype`*`} {-keystore` *`keystore`*`} [-storepass` *`storepass`*`] {-providerName` *`provider_name`*`} {-providerClass` *`provider_class_name`* `{-providerArg` *`provider_arg`*`}} {-v} {-protected} {-J`*`javaoption`*`}`
Reads the certificate or certificate chain (where the latter is supplied in a PKCS#7 formatted reply or a sequence of X.509 certificates) from the file *cert_file*, and stores it in the keystore entry identified by *alias*. If no file is given, the certificate or certificate chain is read from stdin.

keytool 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:

1. to add it to the list of trusted certificates, or
2. to import a certificate reply received from a CA as the result of submitting a Certificate Signing Request (see the -certreq command) to that CA.

Which type of import is intended is indicated by the value of the `-alias` option:

1. **If the alias does not point to a key entry**, then **keytool** assumes you are adding a trusted certificate entry. In this case, the alias should not already exist in the keystore. If the alias does already exist, then **keytool** outputs an error, since there is already a trusted certificate for that alias, and does not import the certificate.
2. **If the alias points to a key entry**, then **keytool** assumes you are importing a certificate reply.

Importing a New Trusted Certificate

Before adding the certificate to the keystore, **keytool** tries to verify 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 has been specified, additional certificates are considered for the chain of trust, namely the certificates in a file named "cacerts".

If **keytool** 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), the certificate information is printed out, and the user is prompted to verify it, e.g., by comparing the displayed certificate fingerprints with the fingerprints obtained from some other (trusted) source of information, which might be the certificate owner himself/herself. Be very careful to ensure the certificate is valid prior to importing it as a "trusted" certificate! -- see WARNING Regarding Importing Trusted Certificates. The user then has the option of aborting the import operation. If the `-noprompt` option is given, however, there will be no interaction with the user.

Importing a Certificate Reply

When importing a certificate reply, the certificate reply is validated using trusted certificates from the keystore, and optionally using the certificates configured in the "cacerts" keystore file (if the `-trustcacerts` option was specified).

The methods of determining whether the certificate reply is trusted are described in the following:

- **If the reply is a single X.509 certificate**, **keytool** 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 used to authenticate the certificate reply form the new certificate chain of *alias*. If a trust chain cannot be established, the certificate reply is not imported. In this case, **keytool** does not print out the certificate and prompt the user to verify it, because it is very hard (if not impossible) 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**, 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 `-trustcacerts` option was specified, **keytool** will attempt to match it with any of the trusted certificates in the keystore or the "cacerts" keystore file. If the chain does not end with a self-signed root CA certificate and the `-trustcacerts` option was specified, **keytool** will try 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 is not found and `-noprompt` option is not specified, the information of the last certificate in the chain is printed out, 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 under *alias*, the old certificate chain is replaced with the new certificate chain in the reply. The old chain can only be replaced if a valid *keypass*, 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 previous releases. This old name is still supported in this release and will be supported in future releases, but for clarify the new name, `-importcert`, is preferred going forward.

`**-importkeystore** -srckeystore *srckeystore* -destkeystore *destkeystore* {-srcstoretype *srcstoretype*} {-deststoretype`

`*deststoretype*} [-srcstorepass *srcstorepass*] [-deststorepass *deststorepass*] {-srcprotected} {-destprotected} {-srcalias *srcalias* {-destalias *destalias*} [-srckeypass *srckeypass*] [-destkeypass *destkeypass*] } {-noprompt} {-srcProviderName *src_provider_name*} {-destProviderName *dest_provider_name*} {-providerClass *provider_class_name* {-providerArg *provider_arg*}} {-v} {-protected} {-J*javaoption*}`

Imports a single entry or all entries from a source keystore to a destination 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 is not provided with *destalias*, then *srcalias* is used as the destination alias. If the source entry is protected by a password, *srckeypass* will be used to recover the entry. If *srckeypass* is not provided, then **keytool** will attempt to use *srcstorepass* to recover the entry. If *srcstorepass* is either not provided or is incorrect, the user will be prompted for a password. The destination entry will be protected using *destkeypass*. If *destkeypass* is not provided, the destination entry will be protected with the source entry password.

If the *srcalias* option is not provided, then all entries in the source keystore are imported into the destination keystore. Each destination entry will be stored under the alias from the source entry. If the source entry is protected by a password, *srcstorepass* will be used to recover the entry. If *srcstorepass* is either not provided or is incorrect, the user will be prompted for a password. If a source keystore entry type is not supported in the destination keystore, or if an error occurs while storing an entry into the destination keystore, the user will be prompted whether to skip the entry and continue, or to quit. The destination entry will be protected with the source entry password.

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

Note that if `-noprompt` is provided, the user will not be prompted for a new destination alias. Existing entries will automatically be overwritten with the destination alias name. Finally, entries that can not be imported are automatically skipped and a warning is output.

`-printcertreq` `{-file *file*}`

Prints the content of a PKCS #10 format certificate request, which can be generated by the keytool -certreq command. The command reads the request from file; if omitted, from the standard input.

Exporting Data

`-certreq {-alias alias} {-dname dname} {-sigalg sigalg} {-file certreq_file} [-keypass keypass] {-storetype storetype} {-keystore keystore} [-storepass storepass] {-providerName provider_name} {-providerClass provider_class_name {-providerArg provider_arg}} {-v} {-protected} {-Jjavaoption}`
Generates a Certificate Signing Request (CSR), using the PKCS#10 format.

A CSR is intended to be sent to a certificate authority (CA). The CA will authenticate the certificate requestor (usually off-line) and will return a certificate or certificate chain, used to replace the existing certificate chain (which initially consists of a self-signed certificate) in the keystore.

The private key associated with *alias* is used to create the PKCS#10 certificate request. In order to access the private key, the appropriate password must be provided, since private keys are protected in the keystore with a password. If *keypass* is not provided at the command line, and is different from the password used to protect the integrity of the keystore, the user is prompted for it. If dname is provided, it's used as the subject in the CSR. Otherwise, the X.500 Distinguished Name associated with alias is used.

sigalg specifies the algorithm that should be used to sign the CSR.

The CSR is stored in the file *certreq_file*. If no file is given, the CSR is output to stdout.

Use the *importcert* command to import the response from the CA.

`-exportcert {-alias alias} {-file cert_file} {-storetype storetype} {-keystore keystore} [-storepass storepass] {-providerName provider_name} {-providerClass provider_class_name {-providerArg provider_arg}} {-rfc} {-v} {-protected} {-Jjavaoption}`
Reads (from the keystore) the certificate associated with *alias*, and stores it in the file *cert_file*.

If no file is given, the certificate is output to stdout.

The certificate is by default output in binary encoding, but will instead be output in the printable encoding format, as defined by the Internet RFC 1421 standard, if the `-rfc` option is specified.

If *alias* refers to a trusted certificate, 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*.

This command was named `-export` in previous releases. This old name is still supported in this release and will be supported in future releases, but for clarify the new name, `-exportcert`, is preferred going forward.

Displaying Data

`**-list** {-alias *alias*} {-storetype *storetype*} {-keystore *keystore*} [-storepass *storepass*] {-providerName *provider_name*} {-providerClass *provider_class_name* {-providerArg *provider_arg*}} {-v | -rfc} {-protected} {-J*javaoption*}`
Prints (to stdout) the contents of the keystore entry identified by *alias*. If no alias is specified, the contents of the entire keystore are printed.

This command by default prints the SHA1 fingerprint of a certificate. If the `-v` option is specified, 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, certificate contents are printed using the printable encoding format, as defined by the Internet RFC 1421 standard

You cannot specify both `-v` and `-rfc`.

`**-printcert** {-file *cert_file* | -sslserver host[:port]} {-jarfile *JAR_file* {-rfc} {-v} {-J*javaoption*}`
Reads the certificate from the file *cert_file*, the SSL server located at *host:port*, or the signed JAR file *JAR_file* (with the option `-jarfile` and prints its contents in a human-readable format. When no port is specified, the standard HTTPS port 443 is assumed. Note that `-sslserver` and `-file` options cannot be provided at the same time. Otherwise, an error is reported. If neither option is given, the certificate is read from stdin.

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

If the certificate is read from a file or stdin, it may be either binary encoded or in printable encoding format, as defined by the Internet RFC 1421 standard

If the SSL server is behind a firewall, `-J-Dhttps.proxyHost=proxyhost` and `-J-Dhttps.proxyPort=proxyport` can be specified on the command line for proxy tunneling. See the JSSE Reference Guide for more information.

Note: This option can be used independently of a keystore.

`-printcrl` `-file` *`crl_`* `{-v}`
Reads the certificate revocation list (CRL) from the file *crl_file*.

A Certificate Revocation List (CRL) is a list of digital certificates which have been revoked by the Certificate Authority (CA) that issued them. The CA generates *crl_file*.

Note: This option can be used independently of a keystore.

Managing the Keystore

`-storepasswd` `[-new` *`new_storepass`*`] {-storetype` *`storetype`*`} {-keystore` *`keystore`*`} [-storepass` *`storepass`*`] {-providerName` *`provider_name`*`} {-providerClass` *`provider_class_name`* `{-providerArg` *`provider_arg`*`}} {-v} {-J`*`javaoption`*`}`
Changes the password used to protect the integrity of the keystore contents. The new password is *new_storepass*, which must be at least 6 characters long.

`-keypasswd` `{-alias` *`alias`*`} [-keypass` *`old_keypass`*`] [-new` *`new_keypass`*`] {-storetype` *`storetype`*`} {-keystore` *`keystore`*`} [-storepass` *`storepass`*`] {-providerName` *`provider_name`*`} {-providerClass` *`provider_class_name`* `{-providerArg` *`provider_arg`*`}} {-v} {-J`*`javaoption`*`}`
Changes the password under which the private/secret key identified by *alias* is protected, from *old_keypass* to *new_keypass*, which must be at least 6 characters long.

If the `-keypass` option is not provided at the command line, and the key password is different from the keystore password, the user is prompted for it.

If the `-new` option is not provided at the command line, the user is prompted for it.

`**-delete** [-alias *alias*] {-storetype *storetype*} {-keystore *keystore*} [-storepass *storepass*] {-providerName *provider_name*} {-providerClass *provider_class_name* {-providerArg *provider_arg*}} {-v} {-protected} {-J*javaoption*}`

Deletes from the keystore the entry identified by *alias*. The user is prompted for the alias, if no alias is provided at the command line.

`**-changealias** {-alias *alias*} [-destalias *destalias*] [-keypass *keypass*] {-storetype *storetype*} {-keystore *keystore*} [-storepass *storepass*] {-providerName *provider_name*} {-providerClass *provider_class_name* {-providerArg *provider_arg*}} {-v} {-protected} {-J*javaoption*}`

Move an existing keystore entry from the specified *alias* to a new alias, *destalias*. If no destination alias is provided, the command will prompt for one. If the original entry is protected with an entry password, the password can be supplied via the "-keypass" option. If no key password is provided, the *storepass* (if given) will be attempted first. If that attempt fails, the user will be prompted for a password.

Lists the basic commands and their options.

For more information about a specific command, enter the following, where *command_name* is the name of the command:

```
    keytool -command_name -help
```

EXAMPLES

Suppose you want to create a keystore for managing your public/private key pair and certificates from entities you trust.

Generating Your Key Pair

The first thing you need to do is create a keystore and generate the key pair. You could use a command such as the following:

```
    keytool -genkeypair -dname "cn=Mark Jones, ou=Java, o=Oracle, c=US"
      -alias business -keypass <new password for private key> -keystore /working/mykeystore
```

```
        -storepass <new password for keystore> -validity 180
```

(Please note: This must be typed as a single line. Multiple lines are used in the examples just for legibility purposes.)

This command creates the keystore named "mykeystore" in the "working" directory (assuming it doesn't already exist), and assigns it the password specified by <*new password for keystore*>. It generates a public/private key pair for the entity whose "distinguished name" has a common name of "Mark Jones", organizational unit of "Java", organization of "Oracle" and two-letter country code of "US". It uses the default "DSA" key generation algorithm to create the keys, both 1024 bits long.

It creates a self-signed certificate (using the default "SHA1withDSA" signature algorithm) that includes the public key and the distinguished name information. This certificate will be valid for 180 days, and is associated with the private key in a keystore entry referred to by the alias "business". The private key is assigned the password specified by <*new password for private key*>.

The command could be significantly shorter if option defaults were accepted. As a matter of fact, no options are required; defaults are used for unspecified options that have default values, and you are prompted for any required values. Thus, you could simply have the following:

```
    keytool -genkeypair
```

In this case, a keystore entry with alias "mykey" is created, with a newly-generated key pair and a certificate that is valid for 90 days. This entry is placed in the keystore named ".keystore" in your home directory. (The keystore is created if it doesn't already exist.) You will be prompted for the distinguished name information, the keystore password, and the private key password.

The rest of the examples assume you executed the `-genkeypair` command without options specified, and that you responded to the prompts with values equal to those given in the first `-genkeypair` command, above (for example, a distinguished name of "cn=Mark Jones, ou=Java, o=Oracle, c=US").

Requesting a Signed Certificate from a Certification Authority

So far all we've got is a self-signed certificate. A certificate is more likely to be trusted by others if it is signed by a Certification Authority (CA). To get such a signature, you first generate a Certificate Signing Request (CSR), via the following:

```
    keytool -certreq -file MarkJ.csr
```

This creates a CSR (for the entity identified by the default alias "mykey") and puts the request in the file named "MarkJ.csr". Submit this file to a CA, such as VeriSign, Inc. The CA will authenticate you, the requestor (usually off-line), and then will return a certificate, signed by them, authenticating your public key. (In some cases, they will actually return 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

You need to replace your 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.

Before you import the certificate reply from a CA, you need one or more "trusted certificates" in your keystore or in the `cacerts` keystore file (which is described in importcert command):

- If the certificate reply is a certificate chain, you just need the top certificate of the chain (that is, the "root" CA certificate authenticating that CA's public key).
- If the certificate reply is a single certificate, you need a certificate for the issuing CA (the one that signed it), and if that certificate is not self-signed, you need a certificate for its signer, and so on, up to a self-signed "root" CA certificate.

The "cacerts" keystore file ships with several VeriSign root CA certificates, so you probably won't need to import a VeriSign certificate as a trusted certificate in your keystore. But if you request a signed certificate from a different CA, and a certificate authenticating that CA's public key hasn't been added to "cacerts", you will need to import a certificate from the CA as a "trusted certificate".

A certificate from a CA is usually either self-signed, or signed by another CA (in which case you also need a certificate authenticating that CA's public key). Suppose company ABC, Inc., is a CA, and you obtain a file named "ABCCA.cer" that is purportedly a self-signed certificate from ABC, authenticating that CA's public key.

Be very careful to ensure the certificate is valid prior to importing it as a "trusted" certificate! View it first (using the **keytool** `-printcert` command, or the **keytool** `-importcert` command without the `-noprompt` option), and make sure that the displayed certificate fingerprint(s) match the expected ones. You can call the person who sent the certificate, and compare the fingerprint(s) that you see with the ones that they show (or that a secure public key repository shows). Only if the fingerprints are equal is it guaranteed that the certificate has not been replaced in transit with somebody else's (for example, an attacker's) certificate. If such an attack took place, and you did not check the certificate before you imported it, you would end up trusting anything the attacker has signed.

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

```
    keytool -importcert -alias abc -file ABCCA.cer
```

This creates a "trusted certificate" entry in the keystore, with the data from the file "ABCCA.cer", and assigns the alias "abc" to the entry.

Importing the Certificate Reply from the CA

Once you've imported a certificate authenticating the public key of the CA 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 thereby replace your self-signed certificate with a certificate chain. This chain is the one returned by the CA in response to your request (if the CA reply is a chain), or one constructed (if the CA reply is a single certificate) using the certificate reply and trusted certificates that are already available in the keystore where you import the reply or in the "cacerts" keystore file.

For example, suppose you sent your certificate signing request to VeriSign. You can then import the reply via the following, which assumes the returned certificate is named "VSMarkJ.cer":

```
    keytool -importcert -trustcacerts -file VSMarkJ.cer
```

Exporting a Certificate Authenticating Your Public Key

Suppose you have used the jarsigner tool to sign a Java ARchive (JAR) file. Clients that want to use the file will want to authenticate your signature.

One way they can do this is by first importing your public key certificate into their keystore as a "trusted" entry. You can export the certificate and supply it to your clients. As an example, you can copy your certificate to a file named `MJ.cer` via the following, assuming the entry is aliased by "mykey":

```
    keytool -exportcert -alias mykey -file MJ.cer
```

Given that certificate, and the signed JAR file, a client can use the **jarsigner** tool to authenticate your signature.

Importing Keystore

The command "importkeystore" is used to import an entire keystore into another keystore, which means all entries from the source keystore, including keys and certificates, are all imported to the destination keystore within 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 **keytool** has difficulties recover the private keys or secret keys from the source keystore, it will prompt you for a password. If it detects alias duplication, it will ask you for a new one, you can specify a new alias or simply allow **keytool** to overwrite the existing one.

For example, to import entries from a normal JKS type keystore key.jks into a PKCS #11 type hardware based keystore, you can use the command:

```
  keytool -importkeystore
    -srckeystore key.jks -destkeystore NONE
    -srcstoretype JKS -deststoretype PKCS11
    -srcstorepass <source keystore password> -deststorepass
<destination keystore 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 see in the above example, you need to specify the alias you want to import. With the srcalias option given, you can also specify the destination alias name in the command line, as well as protection password for a secret/private key and the destination protection password you want. The following command demonstrates this:

```
  keytool -importkeystore
    -srckeystore key.jks -destkeystore NONE
    -srcstoretype JKS -deststoretype PKCS11
    -srcstorepass <source keystore password> -deststorepass
<destination keystore password>
    -srcalias myprivatekey -destalias myoldprivatekey
    -srckeypass <source entry password> -destkeypass <destination
entry password>
    -noprompt
```

Generating Certificates for a Typical SSL Server

The following are keytool commands to generate keypairs and certificates for three entities, namely, Root CA (root), Intermediate CA (ca), and SSL server (server). Ensure that you store all the certificates in the same keystore. In these examples, it is recommended that you specify RSA as the key algorithm.

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

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

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

keytool -storepass <storepass> -keystore server.jks -certreq -
alias server | keytool -storepass <storepass> -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
```

TERMINOLOGY and WARNINGS

KeyStore

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

- KeyStore Entries
  Keystores may have different types of entries. The two most applicable entry types for **keytool** include:

  1. **key entries** - each 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. The **keytool** can handle both types of entries, while the **jarsigner** tool only handle the latter type of entry, that is private keys and their associated certificate chains.
  2. **trusted certificate entries** - each contains a single public key certificate belonging to another party. It is called a "trusted certificate" because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the "subject" (owner) of the certificate. The issuer of the certificate vouches for this, by signing the certificate.

- KeyStore Aliases
  All keystore entries (key and trusted certificate entries) are accessed via unique *aliases*.

  An alias is specified when you add an entity to the keystore using the -genseckey command to generate a secret key, -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, suppose you use the alias `duke` to generate a new public/private key pair and wrap the public key into a self-signed certificate (see Certificate Chains) via the following command:

  ```
  keytool -genkeypair -alias duke -keypass dukekeypasswd
  ```

This specifies an initial password of "dukekeypasswd" 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, you use a command like the following:

```
    keytool -keypasswd -alias duke -keypass dukekeypasswd -new newpass
```

This changes the password from "dukekeypasswd" to "newpass".

Please note: A password should not actually 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, you will be 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**) and a GUI-based tool named **Policy Tool** make use of keystore implementations. Since `KeyStore` is publicly available, users can write additional security applications that use it.

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

  Keystore implementations are provider-based. More specifically, the application interfaces supplied by `KeyStore` are implemented in terms of a "Service Provider Interface" (SPI). That is, there is a corresponding abstract `KeystoreSpi` class, also in the `java.security` package, which defines the 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.) Thus, to provide a keystore implementation, clients must implement a "provider" and supply a KeystoreSpi subclass implementation, as described in How to Implement a Provider for the Java Cryptography Architecture.

Applications can choose different *types* of keystore implementations from different providers, using the "getInstance" factory method 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 itself. Keystore implementations of different types are not compatible.

keytool works on any file-based keystore implementation. (It treats the keystore location that is passed to it at the command line as a filename and converts it to a FileInputStream, from which it loads the keystore information.) The **jarsigner** and **policytool** tools, on the other hand, can read a keystore from any location that can be specified using a URL.

For **keytool** and **jarsigner**, you can specify a keystore type at the command line, via the *-storetype* option. For **Policy Tool**, you can specify a keystore type via the "Keystore" menu.

If you don't explicitly specify a keystore type, the tools choose a keystore implementation based simply on the value of the `keystore.type` property specified in the security properties file. The security properties file is called `java.security`, and it resides in the security properties directory, `java.home/lib/security`, where *java.home* is the runtime environment's directory (the `jre` directory in the SDK or the top-level directory of the Java 2 Runtime Environment).

Each tool gets the `keystore.type` value and then examines all the currently-installed providers until it finds one that implements 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 and applets 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.getDefaultType());
```

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

```
keystore.type=jks
```

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 have a provider package that supplies a keystore implementation for a keystore type called "pkcs12", change the line to

```
keystore.type=pkcs12
```

Note: case doesn't matter in keystore type designations. For example, "JKS" would be considered the same as "jks".

Certificate
A **certificate** (also known as a **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.

- **Certificate Terms**

  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* it has been 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, in others it can be anything from a Unix 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's 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.

Basically, 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 have been 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 (for example, businesses) that are trusted to sign (issue) certificates for other entities. It is assumed that CAs will only create valid and reliable certificates, as they are bound by legal agreements. There are many public Certification Authorities, such as VeriSign, Thawte, 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.

Using **keytool**, it is possible to display, import, and export certificates. It is also possible to generate self-signed certificates.

keytool 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 using 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. **keytool** 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 and/or issuer names over time. Most certificate profile documents strongly recommend that names not be reused, and that certificates should not make use of unique identifiers. Version 2 certificates are not widely used.

X.509 Version 3 is the most recent (1996) and supports the notion of extensions, whereby anyone can define an extension and include it in the certificate. Some common extensions in use today 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, e.g. DNS names, Email addresses, IP addresses). Extensions can be marked *critical* to indicate that the extension should be checked and enforced/used. For example, if a certificate has the KeyUsage extension marked critical and set to "keyCertSign" then if this certificate is presented during SSL communication, it should be rejected, as 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 normally a CA. Using this certificate implies trusting the entity that signed this certificate. (Note that 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, if 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, Organizational Unit, Organization, and Country.)

Subject Public Key Information
This is the public key of the entity being named, together with an algorithm identifier which specifies which public key crypto system this key belongs to and any associated key parameters.

- **Certificate Chains**

keytool 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 corresponding to the private key.

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

Later, after a Certificate Signing Request (CSR) has been generated (see the -certreq command) and sent to a Certification Authority (CA), the response from the CA is imported (see -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 (that is, a certificate from the CA authenticating its own public key) and the last certificate in the chain. In other cases, the CA may 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, authenticating the public key of the CA you sent the CSR to. Then, the next certificate in the chain will be a certificate authenticating the second CA's key, and so on, until a self-signed "root" certificate is reached. Each certificate in the chain (after the first) thus 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) also includes the supporting certificate chain, in addition to the issued certificate. Both reply formats can be handled by **keytool**.

The top-level (root) CA certificate is self-signed. However, the trust into the root's public key does not come from the root certificate itself (anybody could generate a self-signed certificate with the distinguished name of say, the VeriSign root CA!), but from other sources like a newspaper. 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 (using the `-printcert` option) and compare the displayed fingerprint with the well-known fingerprint (obtained from a newspaper, the root CA's Web page, etc.).

- **The cacerts Certificates File**

  A certificates file named **"cacerts"** resides in the security properties directory, `java.home/lib/security`, where *java.home* is the runtime environment's directory (the `jre` directory in the SDK or the top-level directory of the Java 2 Runtime Environment).

The "cacerts" file represents a system-wide keystore with CA certificates. System administrators can configure and manage that file using **keytool**, specifying "jks" as the keystore type. The "cacerts" keystore file ships with a default set of root CA certificates; list them with the following command:

```
keytool -list -keystore java.home/lib/security/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.

IMPORTANT: Verify Your `cacerts` File: Since 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 JRE installation directory. Contact your system administrator if you do not have permission to edit this file.

- **The 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 "Base 64 encoding", facilitates exporting 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, if the `-rfc` option is specified.

  The `-list` command by default prints the SHA1 fingerprint of a certificate. If the `-v` option is specified, the certificate is printed in human-readable format, while if the `-rfc` option is specified, the certificate is output in the printable encoding format.

In its printable encoding format, the encoded certificate is bounded at the beginning by

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

and at the end by

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

X.500 Distinguished Names

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

- *commonName* - common name of a person, e.g., "Susan Jones"
- *organizationUnit* - small organization (e.g., department or division) name, e.g., "Purchasing"
- *organizationName* - large organization name, e.g., "ABCSystems, Inc."
- *localityName* - locality (city) name, e.g., "Palo Alto"
- *stateName* - state or province name, e.g., "California"
- *country* - two-letter country code, e.g., "CH"

When supplying a distinguished name string as the value of a `-dname` option, 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
```

where all the italicized items represent actual values and the above 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
```

and a sample command using such a string is

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

Case does not 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 is not necessary to have all the subcomponents. You may use a subset, for example:

```
CN=Steve Meier, OU=Java, O=Oracle, C=US
```

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

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

It is never necessary to specify a distinguished name string on a command line. If it 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 does not need to be escaped by a "\".

WARNING Regarding Importing Trusted Certificates

IMPORTANT: Be sure to check a certificate very carefully before importing it as a trusted certificate!

View it first (using the `-printcert` command, or the `-importcert` command without the `-noprompt` option), and make sure that the displayed certificate fingerprint(s) match the expected ones. For example, suppose someone sends or emails you a certificate, and 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 in

```
  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 Sep 25 18:01:13 PDT 1997 until: Wed Dec 24
17:01:13 PST 1997
    Certificate Fingerprints:
         MD5:  11:81:AD:92:C8:E5:0E:A2:01:2E:D4:7A:D7:5F:07:6F
         SHA1:
20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:13:0E:5E:FE
         SHA256: 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 fingerprint(s) that you see with the ones that they show. Only if the fingerprints are equal is it guaranteed that the certificate has not been replaced in transit with somebody else's (for example, an attacker's) certificate. If such an attack took place, and you did not check the certificate before you imported it, you would end up trusting anything the attacker has signed (for example, a JAR file with malicious class files inside).

Note: it is not required that you execute a `-printcert` command prior to importing a certificate, since before adding 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 then have the option of aborting the import operation. Note, however, this is only the case if you invoke the `-importcert` command without the `-noprompt` option. If the `-noprompt` option is given, there is no interaction with the user.

Warning Regarding Passwords

Most commands operating 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, respectively). However, a password should not 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, you will be prompted for it.

Warning Regarding Certificate Conformance

The Internet standard RFC 5280 has defined a profile on conforming X.509 certificates, which includes what values and value combinations are valid for certificate fields and extensions. **keytool** has not enforced all these rules so it can generate certificates which do not conform to the standard, and these certificates might be rejected by JRE or other applications. Users should make sure that they provide the correct options for `-dname`, `-ext`, etc.

Commands deemed obsolete and no longer documented: