**10 Steps To Publish Your First Android App On Google Play Store**

Google Play is the premier store for distributing an Android app. After sweating out hours on creating a perfect app, you are now ready to put it out in front of over 1 billion active Android users in more than 190 countries and territories around the world. And, to make the process of publishing your app as smooth as it can be on Google Play Store, here is a quick step by step guide. So, are you ready to publish your android app on Google Play Store?

1. **Create an account**

To publish your app in the Google Play Store, you need to have an account with Google. You might already have a personal email account with them, but it is better to have a separate one to manage your app(s). While registering your publisher account, you would have to pay a registration fee of 25 USD, using Google payments. After this, a verification mail would be sent to you; and then, you can sign in to your Developer Console, where all the action would take place.

1. **Familiarise yourself with Developer Console**

Google Play Developer Console is the starting point and the main dashboard for app publishing operations and tools. Before you go ahead get to know its interface, also familiarise yourself with the list of developer countries and merchant countries. The list of developer countries will tell you about all the locations where distribution to Google Play users is supported. And, if you want to sell apps, or have subscriptions or in-app purchases, then you need to review the list of merchant countries. Apart from this, take a look at the Developer Distribution Agreement, to avoid anything that violates the Google Play’s terms and conditions.

1. **Fill in the necessary account details**

After this, log in to complete your account details. For example, you need to provide your Developer Name, the name which would be displayed in the Google Play Store. After filling in the details, you will have to wait for anything between just a little and 48 (the official time frame) hours, for the Google Play Developer registration to be processed.

1. **Link your merchant account**

If you have a paid app, or one with subscriptions or in-app purchases, then you need to link your Google payments merchant account to your developer profile. Your linked account can be used for financial and tax identification as well as monthly payouts from sales.

1. **Upload your app**

When you are logged in to the Google Play Developer Console, click on “Add New Application” in the “All Applications” tab. Select the correct “Default Language” from the drop-down menu and then type the “Title” of the app, which is your app’s name with which it will appear in the Play Store. After this, select the “Upload APK” button to land on a new page, which would be the homepage for your app. Here, upload the pre-prepared “.apk” file of your app. The timeframe for uploading of the file will depend on its size. The app will remain in the drafts until and unless you publish it.

1. **Alpha and Beta testing the app**

Before you launch your app, it is essential to test it with a sample of end users to get their feedback; and the Google Play takes care of that as well. In the “APK” section of the Developer Console, you will find the options related to “Beta Testing” and “Alpha Testing”. After you have uploaded your app’s “.apk” file(s), you can use these options to receive a URL that can be shared with the testers. Using this link, testers can download your app’s alpha or beta version. Just remember, your testers cannot provide feedback and reviews on the app page; for this, you would have to get in touch with them yourself. You can use their feedback to optimise your app and make relevant changes in it before publishing it in the store.

1. **Provide details for store listing**

Now comes the most important part of uploading the app, as this is the place that will determine how it would be seen in Google Play. After uploading the “.apk” file of your app, go to the “Store Listing” tab. Over there, you need to add the details of the app, like a “Short description” (of 12 to 80 characters) and a “Full description” (of up to 100 characters) of the app. Along with this, add screenshots, a link of the promo video (if you have one), contact details (website name, email id, phone number, etc.), categorisation (application type category and content rating), and other such important details related to the app. After completing the relevant fields, press the “Save” button. You can update your store listing at any time, so do not panic if you have made some mistake or left out a field while filling up.

1. **Add pricing and distribution details**

Now, move to the next tab, which is “Pricing & Distribution” and select whether it is a “Free” or a “Paid” app. Also, select the distribution countries and check the boxes stating that your app complies with the content guidelines. If your app is an educational one, then you can put it in the limelight using “Google Play for Education” option or if it is compatible with an Android TV, then you can add a Leanback launch over here. Once you are done, save the changes and move on to the next step.

1. **Publishing the application**

When all the three tabs- “APK”, “Store Listing” and “Pricing & Distribution”- have been filled and there appears a green check mark next to them, you are all ready to publish the app in Google Play. All you need to do is, click the “Publish this app” button under the “Ready to Publish” drop-down menu at the top right corner of the Developer Console. After you have hit the button, a confirmation bar would show up stating that your app would appear shortly in the Google Play Store. Once your app is published, you can update it as often as you want. You can even make changes with the pricing, configuration and distribution options at any time.

1. **Device Filtering option**

These are a series of extra options that might not seem to be of much importance as you publish the app, but they can prevent your app from getting negative feedbacks. Through Google Play, you can control the distribution of your app according to the device features that are compatible with it. Allow only those devices to find your app in the Play Store that are compatible with your app. There is also an option to manually filter problematic or non-compatible devices, so make the most of it to stay on the top and filter out any negativities.

# Package Your Cordova App for Publishing to an App Store

When you build an application for deployment to an emulator, simulator, physical device or the Cordova Simulate browser, you're building a version of the application specifically crafted for local testing of the app. The app is usually built with debug information packaged into the executable, and the app is signed with a signing key which allows it to work on your local device, but not on any device.

Before you can build your application for deployment to any device through a public app store, you must first configure Visual Studio with the information it needs to complete the process of packaging and signing the application for deployment through an app store. In this article, you'll learn how to configure a project for deployment and create a deployment package for each target platform:

* [Android](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#android)
* [iOS](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#ios)
* [Windows](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#windows)

## Android Applications

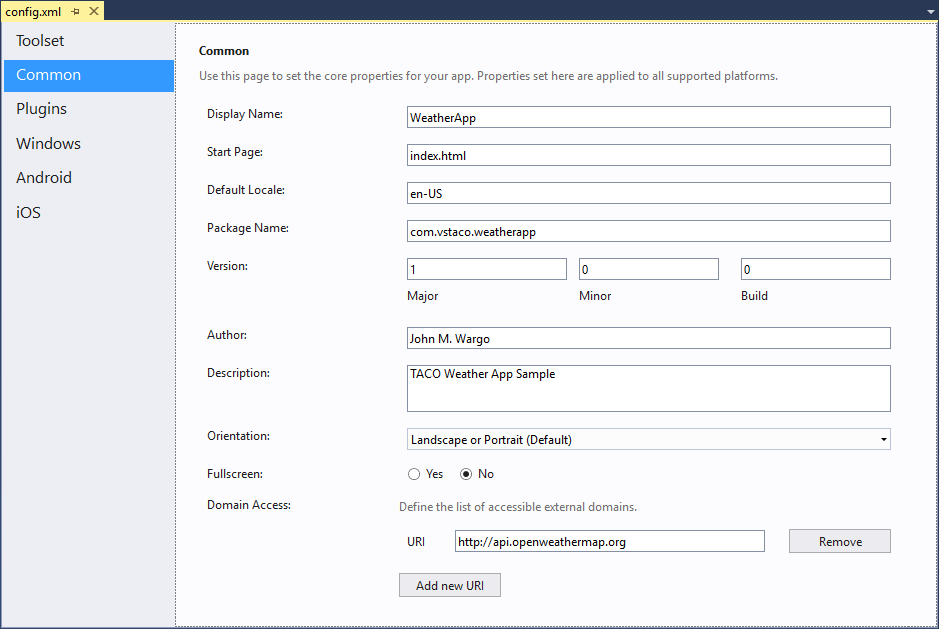
For Android applications, preparing your environment for publishing, and publishing the app requires that you complete the following steps:

1. [Modify Application Settings](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#android-1)
2. [Generate a Private Certificate](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#android-2)
3. [Modify the Android App Build Configuration](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#android-3)
4. [Create the Deployment Package](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#android-4)

### 1. Android: Modify Application Settings

As you prepare to publish your shiny new Cordova application, start in the application's configuration and make sure your settings for the application are correct. A Cordova app's settings are maintained in the project's config.xml file.

1. In the Visual Studio Solution Explorer, double-click the config.xml file to open the custom configuration editor shown in the following figure:

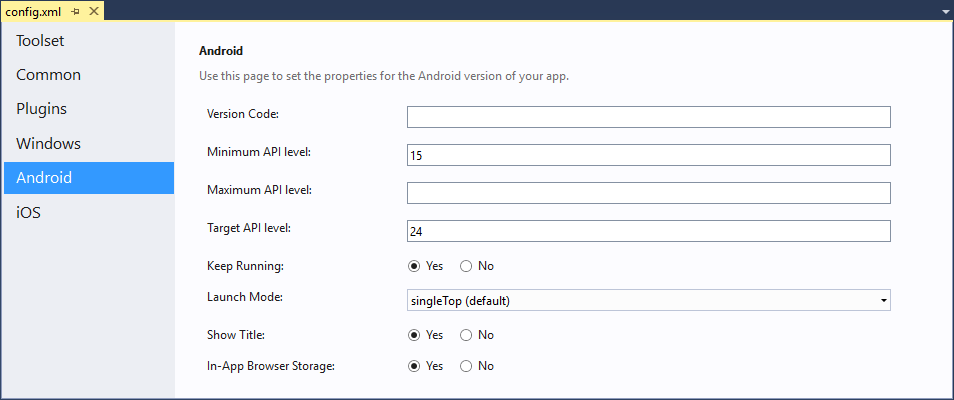


The editor's **Common** tab contains general settings for your app; populate the fields in the form with the appropriate values for your application. The critical settings for any Cordova application are:

* + **Display Name**: the application's public name; this is the how the app will appear in the app store and on the target mobile device's home screen and application listing. Populate this field with a brief word or phrase that describes your app, keeping in mind that app tiles on a device's home screen don't leave much room for text. The value you enter here is added to the config.xml in the <name> element.
  + **Package Name**: the unique identifier for this application. Developers typically populate this field with a combination of the developer's company domain in [**reverse domain name notation**](https://en.wikipedia.org/wiki/Reverse_domain_name_notation) plus the short name for the application. The value you provide here is added to the widget element's id attribute as shown in the following example"
  + **Domain Access**: Manages a list of domains that the application can access; the values you enter here are added as access elements to the config.xml as shown in the following example:

The purpose of most other settings clear from the title, but you can find more information about them here: [The config.xml File](http://cordova.apache.org/docs/en/latest/config_ref/index.html).

1. Switch to the editor's **Android** tab to set Android-specific settings for the application. These settings control the conditions under which the application runs on an android device.



Each input field on the form corresponds to a specific entry in the project's config.xml file:

* + **Version Code**: android-versionCode - a string value, used to set the project's [version code](https://developer.android.com/studio/publish/versioning.html) for the app. See [setting the version code](http://cordova.apache.org/docs/en/latest/guide/platforms/android/index.html#setting-the-version-code) for additional information.
  + **Minimum API level**: android-minSdkVersion - an integer value that represents the minimum device API level for the application. The application will not install on a device if its OS API level is lower than this value.
  + **Maximum API level**: android-maxSdkVersion - an integer value that represents the maximum device API level for the application. The application will not install on a device if its OS API level is higher than this value.
  + **Target API level**: android-targetSdkVersion - an integer value representing the application's [API compatibility](https://developer.android.com/guide/topics/manifest/uses-sdk-element.html).
  + **Keep Running**: KeepRunning - a Boolean value used to determine whether the application stays running in the background after the pause event fires. A false value does not kill the app after a pause event, but simply halts execution of code within the Cordova WebView while the app is in the background.
  + **Lanch Mode**: AndroidLaunchMode - a string value that sets the Activity android:launchMode attribute in the application. This changes what happens when the app is launched from app icon or intent and is already running. Valid values are **standard**, **singleTop**, **singleTask**, and **singleInstance**.
  + **Show Title**: ShowTitle - a Boolean value that controls whether the application displays the app title at the top of the application's main screen.
  + **In-App Browser Storage**: InAppBrowserStorageEnabled - Controls whether pages opened within an InAppBrowser window can access the same localStorage and WebSQL storage as pages opened with the default browser.

You can read about each configuration option in the Cordova [config.xmlreference](http://cordova.apache.org/docs/en/latest/config_ref/index.html#preference) guide.

### 2. Android: Generate a Private Certificate

When running Android applications using the Android SDK (which Visual Studio uses under the covers), applications are signed with a **debug certificate generated by the Android SDK tools**. Before you can sign Android applications for deployment via other means, you must use a signing certificate for your organization. If you already have a certificate you would like to use to sign your Android applications, feel free to skip the remainder of this section.

##### **Note**

Certificates are stored in a **keystore**, if you already have a keystore on your system you'd like to use to store your certificate, you'll need the keystore location and credentials before you continue.

To create a signing certificate, complete the following steps:

1. Open a Windows Command Prompt.

##### **Note**

If your existing keystore is in a protected folder (like c:\ for example), or you'll be generating a keystore in a protected folder, you'll need to open the command prompt in Administrator mode for these steps to complete successfully.

1. If your system is configured with the Java SDK bin folder on the system PATH, then skip to the next step. You can confirm this by typing javac in the command window and pressing enter. If you receive an error message, the JDK is **not** on the path. If you see the Java Compiler help page, then you're in good shape and can skip this step.

In the Command Prompt, change directories to the Java SDK's bin folder. If your development system has the %JAVA\_HOME% environment variable set, then it should be %JAVA\_HOME%\bin. You can also switch to the SDK folder using the complete path (such as: C:\Program Files\Java\jdk1.8.0\_111\bin).

1. In the Command Prompt, execute the following command:

Copy

keytool -genkeypair -v -keystore FILE-PATH\MY-KEYSTORE-NAME.keystore -alias MY-ALIAS -keyalg RSA -keysize 2048 -validity 10000

Replacing FILE-PATH\MY-KEYSTORE-NAME.keystore and MY-ALIAS with the appropriate values for your needs. If you have an existing keystore you want to use, substitute your keystore file path for DRIVE:\FILE-PATH\MY-KEYSTORE-NAME.keystore in the example

For example, if you wanted to write the keystore to the system's dev folder in a keystore file called jd-release-key.keystore and using an alias of android-key for the generated key, you would issue the following command:

Copy

keytool -genkeypair -v -keystore c:\dev\jd-release-key.keystore -alias android-key -keyalg RSA -keysize 2048 -validity 10000

The Java keytool application will launch and prompt you for a series of values it needs to create the keystore and generate the keypair:

Copy

Enter keystore password:

Re-enter new password:

What is your first and last name?

[Unknown]: Joe Developer

What is the name of your organizational unit?

[Unknown]: NA

What is the name of your organization?

[Unknown]: Company

What is the name of your City or Locality?

[Unknown]: Charlotte

What is the name of your State or Province?

[Unknown]: NC

What is the two-letter country code for this unit?

[Unknown]: US

Is CN=Joe Developer, OU=NA, O=Company, L=Charlotte, ST=NC, C=US correct?

[no]: y

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 10,000 days

for: CN=Joe Developer, OU=NA, O=Company, L=Charlotte, ST=NC, C=US

Enter key password for <android-key>

(RETURN if same as keystore password):

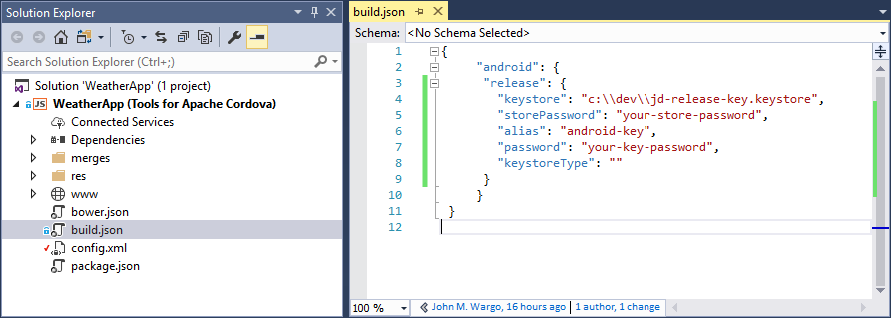
[Storing c:\dev\jd-release-key.keystore]

If you want more detail about this process, see the Android developer documentation here: [Signing your applications](http://developer.android.com/tools/publishing/app-signing.html).

### 3. Android: Modify the Android App Build Configuration

Now that you have a keystore and a signing certificate, you must configure your project to use them.

1. In **Solution Explorer**, expand the project folder, and double-click on the project's build.json file. The build.json file opens in the code editor:



##### **Note**

If the build.json file is missing from your project, its likely that your project was created with an earlier version of Apache Cordova; you should create that file manually (and populate it with content shown in step 2).

1. Populate the build.json file with the keystore and key details:

JavaScriptCopy

{

"android": {

"release": {

"keystore":"c:\\dev\\jd-release-key.keystore",

"storePassword":"your-store-password",

"alias":"android-key",

"password":"your-key-password",

"keystoreType":""

}

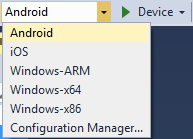
}

}

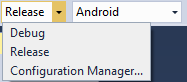
### 4. Android: Create the Deployment Package

The final step involves creating a **Release** build of the Cordova Application.

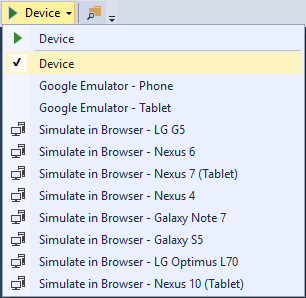
1. On the Standard toolbar, choose the **Android** platform.



1. Choose the **Release** build configuration.



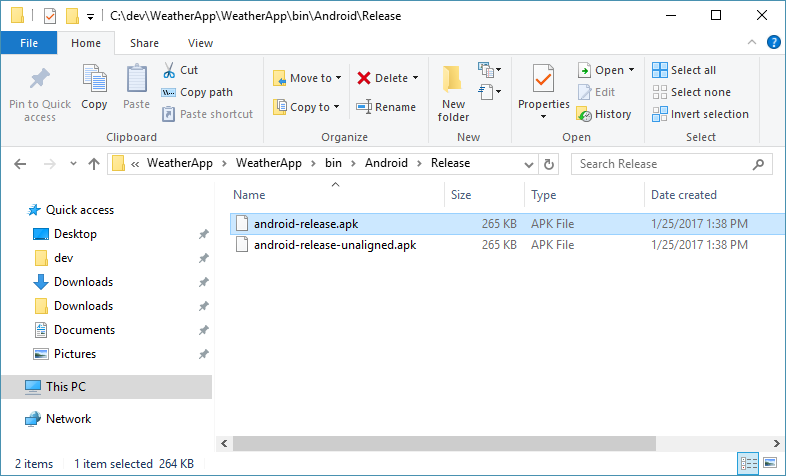
1. Choose one of the Android emulators or a physical device.



##### **Note**

**Do not select** one of the **Simulate in Browser** options, they don't generate a native application binary. Choose only an **Android emulator**or **Device**.

1. In the **Build** menu, select **Build Solution**. This creates a release build of the application, a file with an .apk file extension. This is the file that you'll upload to the store when you deploy the application.
2. When the build completes, look for the .apk file; you'll find it in the project's bin/Android/Release/ folder. When uploading the app to the app store, be sure to select the file that **does not** include the word unaligned in the file name.



At this point, you have a release version of the application all ready to distributed through the Google Play App Store.

## iOS Applications

For iOS applications, preparing your environment for publishing, and publishing the app requires that you complete the following steps:

1. [Request a Distribution Certificate](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#ios-1)
2. [Modify Application Settings](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#ios-2)
3. [Create the Deployment Package](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#ios-3)
4. [Submit Your App to the App Store](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#ios-4)

### 1. iOS: Request a Distribution Certificate

Apple uses **Distribution Certificates** to identify a developer, development team, or organization. You'll need a distribution certificate to deploy applications through the Apple App Store. If your team already has one and you want to use it, refer to [How to share an iOS distribution certificate](http://www.ironpaper.com/webintel/articles/how-to-share-an-ios-distribution-certificate/). Then, skip straight to the [Modify Application Settings](https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store#ios-2) section of this document.

##### **Note**

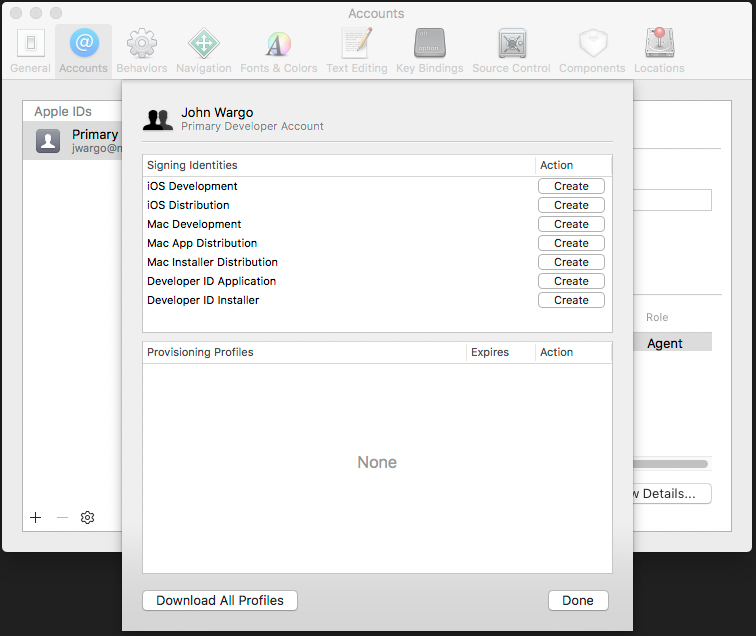
Building iOS applications, requesting distribution certificates, and deploying applications to the app store all require the use of Apple's Xcode development environment running on a computer system running [macOS](http://www.apple.com/macos/). You'll also need an active [Apple iOS developer program account](https://developer.apple.com/ios/). Be sure you have the appropriate iOS development environment setup before continuing.

**To create a distribution certificate, complete the following steps:**

1. Switch to your Macintosh system and open the Xcode IDE.
2. In the system's menu, select **Xcode** -> **Preferences**.

If you haven't done so already, add your developer account Apple ID to the system's configuration. See [Adding an Apple ID to Your Accounts](https://developer.apple.com/library/ios/recipes/xcode_help-accounts_preferences/articles/add_appleid.html) for instructions.

1. In the **Accounts** section, select your developer account, then click the **View Details** button.
2. In the account details window, under **Signing identities**, look for the item labeled **iOS Distribution**. If there is a **Create** button to the right the iOS Distribution entry, click the button to create and download the signing identity. If the Create button isn't shown, that means this step has already been completed.



Looking for more information about signing identities? See [Creating Signing Identities](https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertificates.html#//apple_ref/doc/uid/TP40012582-CH31-SW6) (Optional reading).

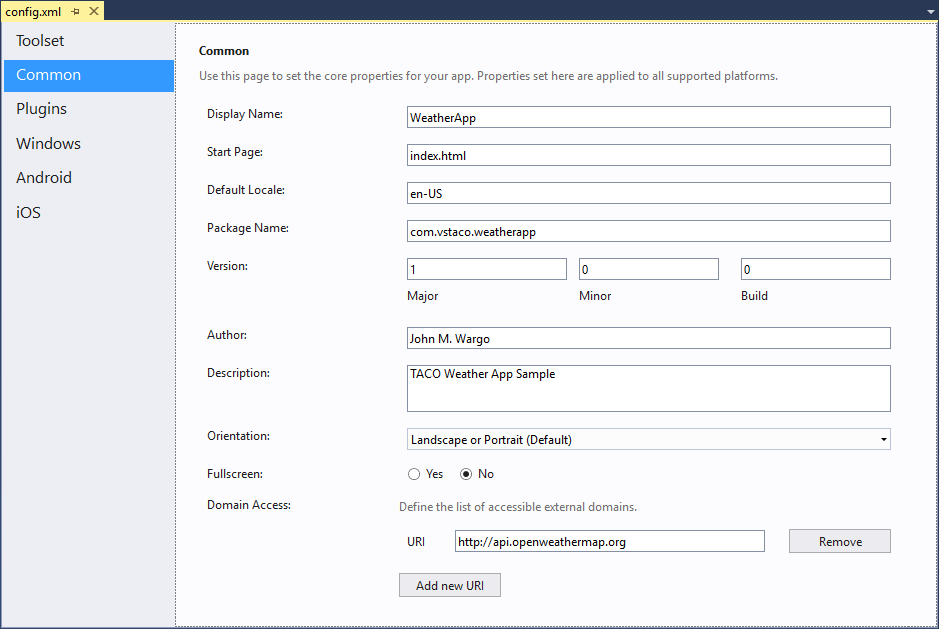
Xcode will submit a request to generate the distribution profile and hide the button you clicked to indicate that the distribution profile exists.

1. Click the **Done** button to close the account details window.

### 2. iOS: Modify Application Settings

As you prepare to publish your shiny new Cordova application, start in the application's configuration and make sure your settings for the application are correct. A Cordova app's settings are maintained in the project's config.xml file.

1. In the Visual Studio Solution Explorer, double-click the config.xml file to open the custom configuration editor shown in the following figure:

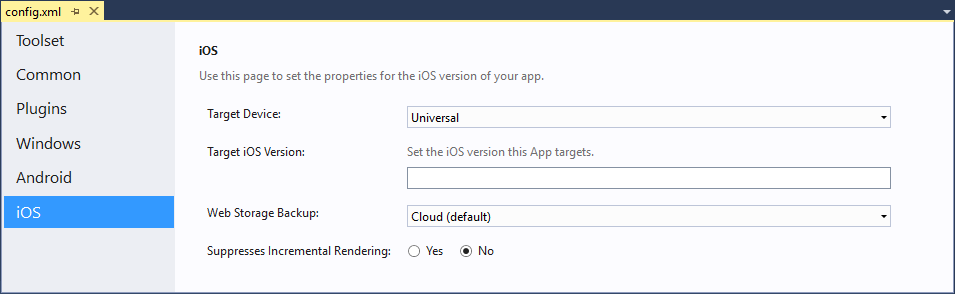


The editor's **Common** tab contains general settings for your app; populate the fields in the form with the appropriate values for your application. The critical settings for any Cordova application are:

* + **Display Name**: the application's public name; this is the how the app will appear in the app store and on the target mobile device's home screen and application listing. Populate this field with a brief word or phrase that describes your app, keeping in mind that app tiles on a device's home screen don't leave much room for text. The value you enter here is added to the config.xml in the <name> element.
  + **Package Name**: the unique identifier for this application. Developers typically populate this field with a combination of the developer's company domain in [**reverse domain name notation**](https://en.wikipedia.org/wiki/Reverse_domain_name_notation) plus the short name for the application. The value you provide here is added to the widget element's id attribute as shown in the following example"
  + **Domain Access**: Manages a list of domains that the application can access; the values you enter here are added as access elements to the config.xml as shown in the following example:

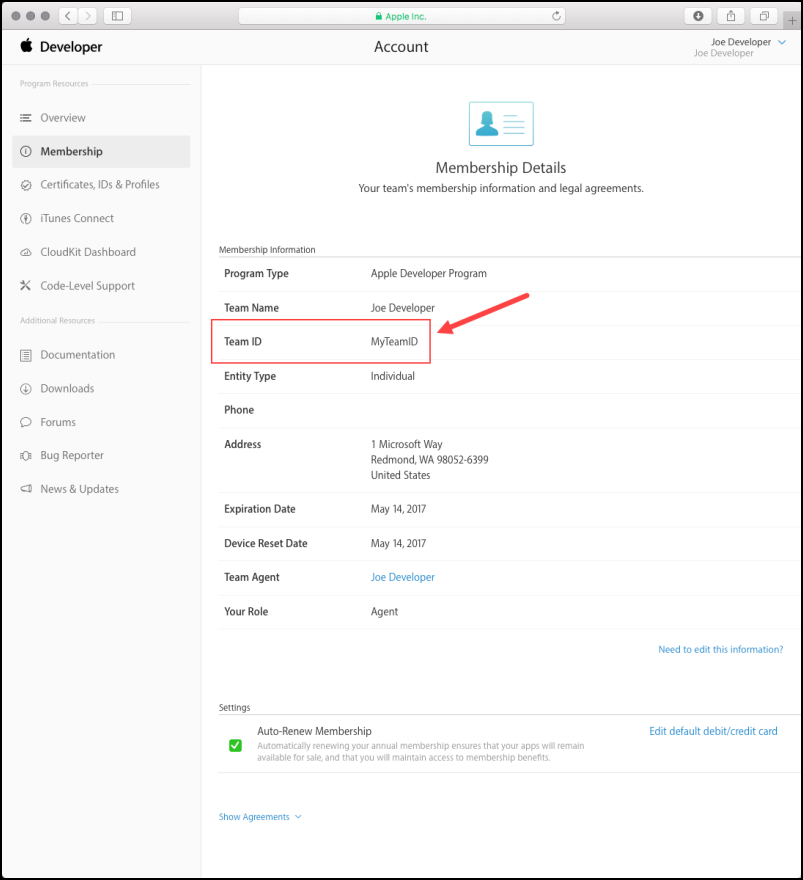
The purpose of most other settings clear from the title, but you can find more information about them here: [The config.xml File](http://cordova.apache.org/docs/en/latest/config_ref/index.html).

1. Switch to the **iOS** tab to set application settings specific to the iOS platform. Populate the fields on the form using the appropriate values for your application:
   * **Target Device**: Used to specify the type of device the application targets; valid options are: **handset**, **tablet**, and **universal**.
   * **Target iOS Version**: Sets the MinimumOSVersion in the generated .ipa file. For more information on this setting, refer to [Configuring a Project for SDK-Based Development](<https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/cross_development/Configuring/configuring.html>).
   * **Web Storage Backup**: Used to specify how iOS backs up application data stored using web storage. Available options are: **none**, **local**, or **cloud**. The cloud option backs up application data using iCloud. The local option enables local backups via iTunes sync. The none option disables storage backup.
   * **Suppress Incremental Rendering**: When set to true, screen rendering is delayed until all content has been received.



1. Even though Xcode manages provisioning and signing, you won't be opening Xcode to configure it for your project, this is all handled behind the scenes by Visual Studio TACO. Instead, you'll configure your **iOS Developer credentials**using the Cordova project's build.json file. During the build process, the Cordova CLI (invoked by Visual Studio TACO) copies settings for the iOS build from the build.json file to the Xcode project's configuration. Next, the remotebuild process (described in detail in the TACO [iOS Setup Guide](https://taco.visualstudio.com/en-us/docs/vs-taco-2017-ios-guide/)) executes the Xcode command-line tools to build and sign the app using the settings you provided.

To configure the iOS build process, you only need one piece of information, the Team ID for your Apple Developer program account. Open your browser of choice and navigate to <https://developer.apple.com/account>. After you've authenticated to the site, open the **Membership** page shown in the following figure. Note the Team ID from the page (it will be a series of letters and numbers, **not** MyTeamID as shown in the figure).



Next, open the Cordova project's build.json file, and add the following JSON object to the file, replacing MyTeamID with the correct value for your account:

JSONCopy

"ios": {

"debug": {

"developmentTeam": "MyTeamID"

},

"release": {

"developmentTeam": "MyTeamID",

"codeSignIdentity": "iPhone Developer",

"packageType": "ad-hoc"

}

}

Leave codeSignIdentity and packageType alone as those values are already set as needed. The data you'll be adding must coencide with the Android project settings that are probably already in the file. The resulting JSON object should look something like the following:

JSONCopy

{

"android": {

"release": {

"keystore":"c:\\dev\\jd-release-key.keystore",

"storePassword":"your-store-password",

"alias":"android-key",

"password":"your-key-password",

"keystoreType":""

}

},

"ios": {

"debug": {

"developmentTeam": "MyTeamID"

},

"release": {

"developmentTeam": "MyTeamID",

"codeSignIdentity": "iPhone Developer",

"packageType": "ad-hoc"

}

}

}

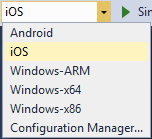
##### **Note**

Don't forget to include the comma between the android and iosobjects.

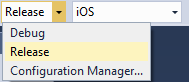
### 3. iOS: Create the Deployment Package

At this point, you're all ready to build the app to generate the package that you will submit to the app store for distribution.

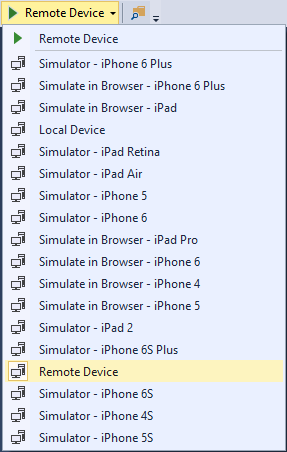
1. On your Macintosh system, make sure that the remote agent is running. Refer to the TACO [iOS Setup Guide](https://taco.visualstudio.com/en-us/docs/vs-taco-2017-ios-guide/) for details.
2. With the Cordova project open in Visual Studio, on the Standard toolbar, choose the **iOS** platform.



1. Choose the **Release** build configuration.

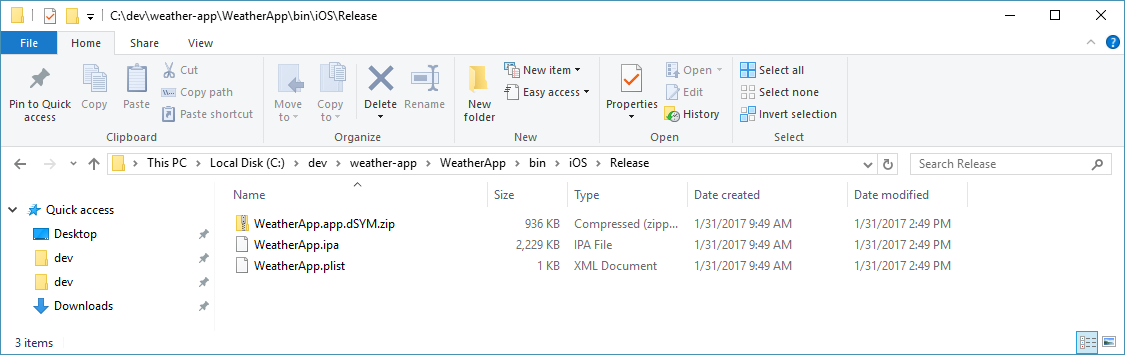


1. For execution target, select **Remote Device**.



1. On the **Build** menu, choose **Build Solution**.

This starts a build on the remotebuild agent and uses the distribution certificate and a newly generated provisioning profile to build a release signed iOS Application Archive (.ipa) file. You can find the resulting file in the bin/iOS/Release folder of your project.



### 4. iOS: Submit Your App to the App Store

1. Copy the application files onto a folder on your Mac.
2. Follow Apple's procedures for submitting the app to the App Store.

# keytool - Key and Certificate Management Tool

Manages a keystore (database) of cryptographic keys, X.509 certificate chains, and trusted certificates.

## SYNOPSIS

**keytool** [ [commands](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#Commands) ]

The keytool command interface has changed in Java SE 6. See the [Changes](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#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*](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#KeyStore).

## COMMAND AND OPTION NOTES

The various commands and their options are listed and described [below](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#Commands) . 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#OptionDefaults) 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#genkeypairCmd))

"DES" (when using [-genseckey](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#genseckeyCmd))

-keysize

1024 (when using [-genkeypair](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#genkeypairCmd))

56 (when using [-genseckey](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#genseckeyCmd) and -keyalg is "DES")

168 (when using [-genseckey](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#genseckeyCmd) 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", and if the underlying private key is of type "RSA", *-sigalg* defaults to "SHA256withRSA". Please consult the [Java Cryptography Architecture API Specification & Reference](https://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#AppA) 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 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#KeyStoreImplementation) 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 *storepass***

The password which is used to protect the integrity of the keystore.

storepass must be at least 6 characters long. It 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.

## COMMANDS

### Creating or Adding Data to the Keystore

**-genkeypair {-alias *alias*} {-keyalg *keyalg*} {-keysize *keysize*} {-sigalg *sigalg*} [-dname *dname*] [-keypass *keypass*] {-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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#DName) 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.

valDays tells the number of days 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 clarify 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) from the file *cert\_file*, and stores it in the keystore entry identified by *alias*. If no file is given, the certificate or PKCS#7 reply 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#EncodeCertificate). 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#certreqCmd) 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"](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#TrustedCertWarning). 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#cacerts) (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**, the chain is first ordered (with the user certificate first and the self-signed root CA certificate last), before **keytool** attempts to match the root CA certificate provided in the reply with any of the trusted certificates in the keystore or the "cacerts" keystore file (if the -trustcacerts option was specified). If no match can be found, the information of the root CA certificate 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 root CA itself. 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.

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.

### Exporting Data

**-certreq {-alias *alias*} {-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} {-J*javaoption*}**

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 and X.500 Distinguished Name associated with alias are 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.

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} {-J*javaoption*}**

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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#EncodeCertificate), 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 MD5 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#EncodeCertificate)

You cannot specify both -v and -rfc.

**-printcert {-file *cert\_file*} {-v} {-J*javaoption*}**

Reads the certificate from the file cert\_file, and prints its contents in a human-readable format. If no file is given, the certificate is read from stdin.

The certificate may be either binary encoded or in printable encoding format, as defined by the [Internet RFC 1421 standard](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#EncodeCertificate).

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.

### Getting Help

**-help**

Lists the basic commands and their options.

## 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=JavaSoft, o=Sun, c=US"

-alias business -keypass kpi135 -keystore C:\working\mykeystore

-storepass ab987c -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 "C:\working" directory (assuming it doesn't already exist), and assigns it the password "ab987c". It generates a public/private key pair for the entity whose "distinguished name" has a common name of "Mark Jones", organizational unit of "JavaSoft", organization of "Sun" 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 "kpi135".

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 -genkeypaircommand, above (a private key password of "kpi135", etc.)

### 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#importCertCmd)):

* 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 five 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's 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html) 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 changeit -deststorepass topsecret

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 desination alias name in the command line, as well as protection password for a secret/private key and the destination protection password you want. In this way, you can issue a **keytool** command that will never ask you a question. This makes it very convenient to include a **keytool** command into a script file, like this:

keytool -importkeystore

-srckeystore key.jks -destkeystore NONE

-srcstoretype JKS -deststoretype PKCS11

-srcstorepass changeit -deststorepass topsecret

-srcalias myprivatekey -destalias myoldprivatekey

-srckeypass oldkeypass -destkeypass mynewkeypass

-noprompt

## 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"](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#CertChains) for the corresponding public key. The **keytool** can handle both types od entry, while **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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#genseckeyCmd) command to generate a secret key, [-genkeypair](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#genkeypairCmd) command to generate a key pair (public and private key) or the [-importcert](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#importCertCmd) 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#CertChains)) via the following command:

keytool -genkeypair -alias duke -keypass dukekeypasswd

This specifies an inital password of "dukekeypasswd" required by subsequent commands to access the private key assocated 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 Sun Microsystems. 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](https://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/HowToImplAProvider.html).

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 keytore 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 jredirectory 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 Sun). 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](http://www.verisign.com/), [Thawte](http://www.thawte.com/), [Entrust](http://www.entrust.com/), and so on. You can also run your own Certification Authority using products such as the Netscape/Microsoft Certificate Servers 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#DName) 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#DName)(DN) of the entity, for example,

CN=Java Duke, OU=Java Software Division, O=Sun Microsystems Inc, 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#genkeypairCmd) 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](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#certreqCmd) command) and sent to a Certification Authority (CA), the response from the CA is imported (see [-importcert](https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html#importCertCmd)), 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 webpage, 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 several root CA certificates with the following aliases and X.500 owner distinguished names:

* + **Alias**: thawtepersonalfreemailca  
    **Owner DN**: EmailAddress=personal-freemail@thawte.com,  
    CN=Thawte Personal Freemail CA,  
    OU=Certification Services Division,  
    O=Thawte Consulting, L=Cape Town, ST=Western Cape, C=ZA
  + **Alias**: thawtepersonalbasicca  
    **Owner DN**: EmailAddress=personal-basic@thawte.com,  
    CN=Thawte Personal Basic CA,  
    OU=Certification Services Division,  
    O=Thawte Consulting, L=Cape Town, ST=Western Cape, C=ZA
  + **Alias**: thawtepersonalpremiumca  
    **Owner DN**: EmailAddress=personal-premium@thawte.com,  
    CN=Thawte Personal Premium CA,  
    OU=Certification Services Division,  
    O=Thawte Consulting, L=Cape Town, ST=Western Cape, C=ZA
  + **Alias**: thawteserverca  
    **Owner DN**: EmailAddress=server-certs@thawte.com,  
    CN=Thawte Server CA, OU=Certification Services Division,  
    O=Thawte Consulting cc, L=Cape Town, ST=Western Cape, C=ZA
  + **Alias**: thawtepremiumserverca  
    **Owner DN**: EmailAddress=premium-server@thawte.com,  
    CN=Thawte Premium Server CA,  
    OU=Certification Services Division,  
    O=Thawte Consulting cc, L=Cape Town, ST=Western Cape, C=ZA
  + **Alias**: verisignclass1ca  
    **Owner DN**: OU=Class 1 Public Primary Certification Authority,  
    O="VeriSign, Inc.", C=US
  + **Alias**: verisignclass2ca  
    **Owner DN**: OU=Class 2 Public Primary Certification Authority,  
    O="VeriSign, Inc.", C=US
  + **Alias**: verisignclass3ca  
    **Owner DN**: OU=Class 3 Public Primary Certification Authority,  
    O="VeriSign, Inc.", C=US
  + **Alias**: verisignserverca  
    **Owner DN**: OU=Secure Server Certification Authority,  
    O="RSA Data Security, Inc.", C=US
  + **Alias**: verisignclass1g2ca  
    **Owner DN**: OU=VeriSign Trust Network,  
    OU="(c) 1998 VeriSign, Inc. - For authorized use only",  
    OU=Class 1 Public Primary Certification Authority - G2,  
    O="VeriSign, Inc.", C=US
  + **Alias**: verisignclass1g3ca  
    **Owner DN**: CN=VeriSign Class 1 Public Primary Certification Authority - G3, OU="(c) 1999 VeriSign, Inc. - For authorized use only",  
    OU=VeriSign Trust Network,  
    O="VeriSign, Inc.", C=US
  + **Alias**: verisignclass2g2ca  
    **Owner DN**: OU=VeriSign Trust Network,  
    OU="(c) 1998 VeriSign, Inc. - For authorized use only",  
    OU=Class 2 Public Primary Certification Authority - G2,  
    O="VeriSign, Inc.", C=US
  + **Alias**: verisignclass2g3ca  
    **Owner DN**: CN=VeriSign Class 2 Public Primary Certification Authority - G3,  
    OU="(c) 1999 VeriSign, Inc. - For authorized use only",  
    OU=VeriSign Trust Network,  
    O="VeriSign, Inc.", C=US
  + **Alias**: verisignclass3g2ca  
    **Owner DN**: OU=VeriSign Trust Network,  
    OU="(c) 1998 VeriSign, Inc. - For authorized use only",  
    OU=Class 3 Public Primary Certification Authority - G2,  
    O="VeriSign, Inc.", C=US
  + **Alias**: verisignclass3g3ca  
    **Owner DN**: CN=VeriSign Class 3 Public Primary Certification Authority - G3,  
    OU="(c) 1999 VeriSign, Inc. - For authorized use only",  
    OU=VeriSign Trust Network,  
    O="VeriSign, Inc.", C=US
  + **Alias**: baltimorecodesigningca  
    **Owner DN**: CN=Baltimore CyberTrust Code Signing Root,  
    OU=CyberTrust, O=Baltimore, C=IE
  + **Alias**: gtecybertrustglobalca  
    **Owner DN**: CN=GTE CyberTrust Global Root,  
    OU="GTE CyberTrust Solutions, Inc.", O=GTE Corporation, C=US
  + **Alias**: baltimorecybertrustca  
    **Owner DN**: CN=Baltimore CyberTrust Root,  
    OU=CyberTrust, O=Baltimore, C=IE
  + **Alias**: gtecybertrust5ca  
    **Owner DN**: CN=GTE CyberTrust Root 5,  
    OU="GTE CyberTrust Solutions, Inc.", O=GTE Corporation, C=US
  + **Alias**: entrustclientca  
    **Owner DN**: CN=Entrust.net Client Certification Authority,  
    OU=(c) 1999 Entrust.net Limited,  
    OU=www.entrust.net/Client\_CA\_Info/CPS incorp. by ref. limits liab.,  
    O=Entrust.net, C=US
  + **Alias**: entrustglobalclientca  
    **Owner DN**: CN=Entrust.net Client Certification Authority,  
    OU=(c) 2000 Entrust.net Limited,  
    OU=www.entrust.net/GCCA\_CPS incorp. by ref. (limits liab.),  
    O=Entrust.net
  + **Alias**: entrust2048ca  
    **Owner DN**: CN=Entrust.net Certification Authority (2048),  
    OU=(c) 1999 Entrust.net Limited,  
    OU=www.entrust.net/CPS\_2048 incorp. by ref. (limits liab.),  
    O=Entrust.net
  + **Alias**: entrustsslca  
    **Owner DN**: CN=Entrust.net Secure Server Certification Authority,  
    OU=(c) 1999 Entrust.net Limited,  
    OU=www.entrust.net/CPS incorp. by ref. (limits liab.),  
    O=Entrust.net, C=US
  + **Alias**: entrustgsslca  
    **Owner DN**: CN=Entrust.net Secure Server Certification Authority,  
    OU=(c) 2000 Entrust.net Limited,  
    OU=www.entrust.net/SSL\_CPS incorp. by ref. (limits liab.),  
    O=Entrust.net
  + **Alias**: godaddyclass2ca  
    **Owner DN**: OU=Go Daddy Class 2 Certification Authority,  
    O="The Go Daddy Group, Inc.", C=US
  + **Alias**: starfieldclass2ca  
    **Owner DN**: OU=Starfield Class 2 Certification Authority,  
    O="Starfield Technologies, Inc.", C=US
  + **Alias**: valicertclass2ca  
    **Owner DN**: EMAILADDRESS=info@valicert.com,  
    CN=http://www.valicert.com/,  
    OU=ValiCert Class 2 Policy Validation Authority,  
    O="ValiCert, Inc.", L=ValiCert Validation Network
  + **Alias**: geotrustglobalca  
    **Owner DN**: CN=GeoTrust Global CA,  
    O=GeoTrust Inc., C=US
  + **Alias**: equifaxsecureca  
    **Owner DN**: OU=Equifax Secure Certificate Authority,  
    O=Equifax, C=US
  + **Alias**: equifaxsecureebusinessca1  
    **Owner DN**: CN=Equifax Secure eBusiness CA-1,  
    O=Equifax Secure Inc., C=US
  + **Alias**: equifaxsecureebusinessca2  
    **Owner DN**: OU=Equifax Secure eBusiness CA-2,  
    O=Equifax Secure, C=US
  + **Alias**: equifaxsecureglobalebusinessca1  
    **Owner DN**: CN=Equifax Secure Global eBusiness CA-1,  
    O=Equifax Secure Inc., C=US
  + **Alias**: soneraclass1ca  
    **Owner DN**: CN=Sonera Class1 CA, O=Sonera, C=FI
  + **Alias**: soneraclass2ca  
    **Owner DN**: CN=Sonera Class2 CA, O=Sonera, C=FI
  + **Alias**: comodoaaaca  
    **Owner DN**: CN=AAA Certificate Services,  
    O=Comodo CA Limited, L=Salford, ST=Greater Manchester, C=GB
  + **Alias**: addtrustclass1ca  
    **Owner DN**: CN=AddTrust Class 1 CA Root,  
    OU=AddTrust TTP Network, O=AddTrust AB, C=SE
  + **Alias**: addtrustexternalca  
    **Owner DN**: CN=AddTrust External CA Root,  
    OU=AddTrust External TTP Network, O=AddTrust AB, C=SE
  + **Alias**: addtrustqualifiedca  
    **Owner DN**: CN=AddTrust Qualified CA Root,  
    OU=AddTrust TTP Network, O=AddTrust AB, C=SE
  + **Alias**: utnuserfirsthardwareca  
    **Owner DN**: CN=UTN-USERFirst-Hardware,  
    OU=http://www.usertrust.com, O=The USERTRUST Network,  
    L=Salt Lake City, ST=UT, C=US
  + **Alias**: utnuserfirstclientauthemailca  
    **Owner DN**: CN=UTN-USERFirst-Client Authentication and Email,  
    OU=http://www.usertrust.com, O=The USERTRUST Network,  
    L=Salt Lake City, ST=UT, C=US
  + **Alias**: utndatacorpsgcca  
    **Owner DN**: CN=UTN - DATACorp SGC,  
    OU=http://www.usertrust.com, O=The USERTRUST Network,  
    L=Salt Lake City, ST=UT, C=US
  + **Alias**: utnuserfirstobjectca  
    **Owner DN**: CN=UTN-USERFirst-Object,  
    OU=http://www.usertrust.com, O=The USERTRUST Network,  
    L=Salt Lake City, ST=UT, C=US

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 MD5 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=JavaSoft, O=Sun, L=Cupertino, S=California, C=US

and a sample command using such a string is

keytool -genkeypair -dname "CN=Mark Smith, OU=JavaSoft, O=Sun, 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=SunSoft, O=Sun, 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, o=Sun Microsystems\, Inc., o=sun, 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 C:\temp\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 C:\temp\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

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 -importcertcommand 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 -importcertcommand 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.