Device Administration

In this document

Key classes

- `DeviceAdminReceiver`
- `DevicePolicyManager`
- `DeviceAdminInfo`

Android 2.2 introduces support for enterprise applications by offering the Android Device Administration API. The Device Administration API provides device administration features at the system level. These APIs allow you to create security-aware applications that are useful in enterprise settings, in which IT professionals require rich control over employee devices. For example, the built-in Android Email application has leveraged the new APIs to improve Exchange support. Through the Email application, Exchange administrators can enforce password policies — including alphanumeric passwords or numeric PINs — across devices. Administrators can also remotely wipe (that is, restore factory defaults on) lost or stolen handsets. Exchange users can sync their email and calendar data.

This document is intended for developers who want to develop enterprise solutions for Android-powered devices. It discusses the various features provided by the Device Administration API to provide stronger security for employee devices that are powered by Android.

> **Note** For information on building a Work Policy Controller for Android for Work deployments, see Build a Device Policy Controller.

Device Administration API Overview

Here are examples of the types of applications that might use the Device Administration API:

- Email clients.
- Security applications that do remote wipe.
- Device management services and applications.

How does it work?

You use the Device Administration API to write device admin applications that users install on their devices. The device admin application enforces the desired policies. Here's how it works:

- A system administrator writes a device admin application that enforces remote/local device security policies. These policies could be hard-coded into the app, or the application could dynamically fetch policies from a third-party server.
- The application is installed on users' devices. Android does not currently have an automated provisioning solution. Some of the ways a sysadmin might distribute the application to users are as follows:

  - Google Play.
  - Enabling installation from another store.
  - Distributing the application through other means, such as email or websites.
- The system prompts the user to enable the device admin application. How and when this happens depends on how the application is implemented.
- Once users enable the device admin application, they are subject to its policies. Complying with those policies typically confers benefits, such as access to sensitive systems and data.

If users do not enable the device admin app, it remains on the device, but in an inactive state. Users will not be subject to its policies, and they will conversely not get any of the application's benefits—for example, they may not be able to sync data.

If a user fails to comply with the policies (for example, if a user sets a password that violates the guidelines), it is up to the application to decide how to handle this. However, typically this will result in the user not being able to sync data.

If a device attempts to connect to a server that requires policies not supported in the Device Administration API, the connection will not be allowed. The Device Administration API does not currently allow partial provisioning. In other words, if a device (for example, a legacy device) does not support all of the stated policies, there is no way to allow the device to connect.

If a device contains multiple enabled admin applications, the strictest policy is enforced. There is no way to target a particular admin application.

To uninstall an existing device admin application, users need to first unregister the application as an administrator.

Policies

In an enterprise setting, it's often the case that employee devices must adhere to a strict set of policies that govern the use of the device. The Device Administration API supports the policies listed in Table 1. Note that the Device Administration API currently only supports passwords for screen lock:

Table 1. Policies supported by the Device Administration API.

Policy	Description
Password enabled	Requires that devices ask for PIN or passwords.
Minimum password length	Set the required number of characters for the password. For example, you can require PIN or passwords to have at least six characters.
Alphanumeric password required	Requires that passwords have a combination of letters and numbers. They may include symbolic characters.
Complex password required	Requires that passwords must contain at least a letter, a numerical digit, and a special symbol. Introduced in Android 3.0.
Minimum letters required in password	The minimum number of letters required in the password for all admins or a particular one. Introduced in Android 3.0.
Minimum lowercase letters required in password	The minimum number of lowercase letters required in the password for all admins or a particular one. Introduced in Android 3.0.
Minimum non-letter characters required in password	The minimum number of non-letter characters required in the password for all admins or a particular one. Introduced in Android 3.0.
Minimum numerical digits required in	The minimum number of numerical digits required in the password for all admins or a particular one. Introduced in Android 3.0.

password	
Minimum symbols required in password	The minimum number of symbols required in the password for all admins or a particular one. Introduced in Android 3.0.
Minimum uppercase letters required in password	The minimum number of uppercase letters required in the password for all admins or a particular one. Introduced in Android 3.0.
Password expiration timeout	When the password will expire, expressed as a delta in milliseconds from when a device admin sets the expiration timeout. Introduced in Android 3.0.
Password history restriction	This policy prevents users from reusing the last *n* unique passwords. This policy is typically used in conjunction with `setPasswordExpirationTimeout()`, which forces users to update their passwords after a specified amount of time has elapsed. Introduced in Android 3.0.
Maximum failed password attempts	Specifies how many times a user can enter the wrong password before the device wipes its data. The Device Administration API also allows administrators to remotely reset the device to factory defaults. This secures data in case the device is lost or stolen.
Maximum inactivity time lock	Sets the length of time since the user last touched the screen or pressed a button before the device locks the screen. When this happens, users need to enter their PIN or passwords again before they can use their devices and access data. The value can be between 1 and 60 minutes.
Require storage encryption	Specifies that the storage area should be encrypted, if the device supports it. Introduced in Android 3.0.
Disable camera	Specifies that the camera should be disabled. Note that this doesn't have to be a permanent disabling. The camera can be enabled/disabled dynamically based on context, time, and so on. Introduced in Android 4.0.

Other features

In addition to supporting the policies listed in the above table, the Device Administration API lets you do the following:

- Prompt user to set a new password.
- Lock device immediately.
- Wipe the device's data (that is, restore the device to its factory defaults).

Sample Application

The examples used in this document are based on the Device Administration API sample, which is included in the SDK samples (available through the Android SDK Manager) and located on your system as `<sdk_root>/ApiDemos/app/src/main/java/com/example/android/apis/app/DeviceAdminSample.java`.

The sample application offers a demo of device admin features. It presents users with a user interface that lets them enable the device admin application. Once they've enabled the application, they can use the buttons in the user interface to do the following:

- Set password quality.
- Specify requirements for the user's password, such as minimum length, the minimum number of numeric characters it must contain, and so on.
- Set the password. If the password does not conform to the specified policies, the system returns an error.
- Set how many failed password attempts can occur before the device is wiped (that is, restored to factory settings).
- Set how long from now the password will expire.
- Set the password history length (*length* refers to number of old passwords stored in the history). This prevents users from reusing one of the last *n* passwords they previously used.

- Specify that the storage area should be encrypted, if the device supports it.
- Set the maximum amount of inactive time that can elapse before the device locks.
- Make the device lock immediately.
- Wipe the device's data (that is, restore factory settings).
- Disable the camera.

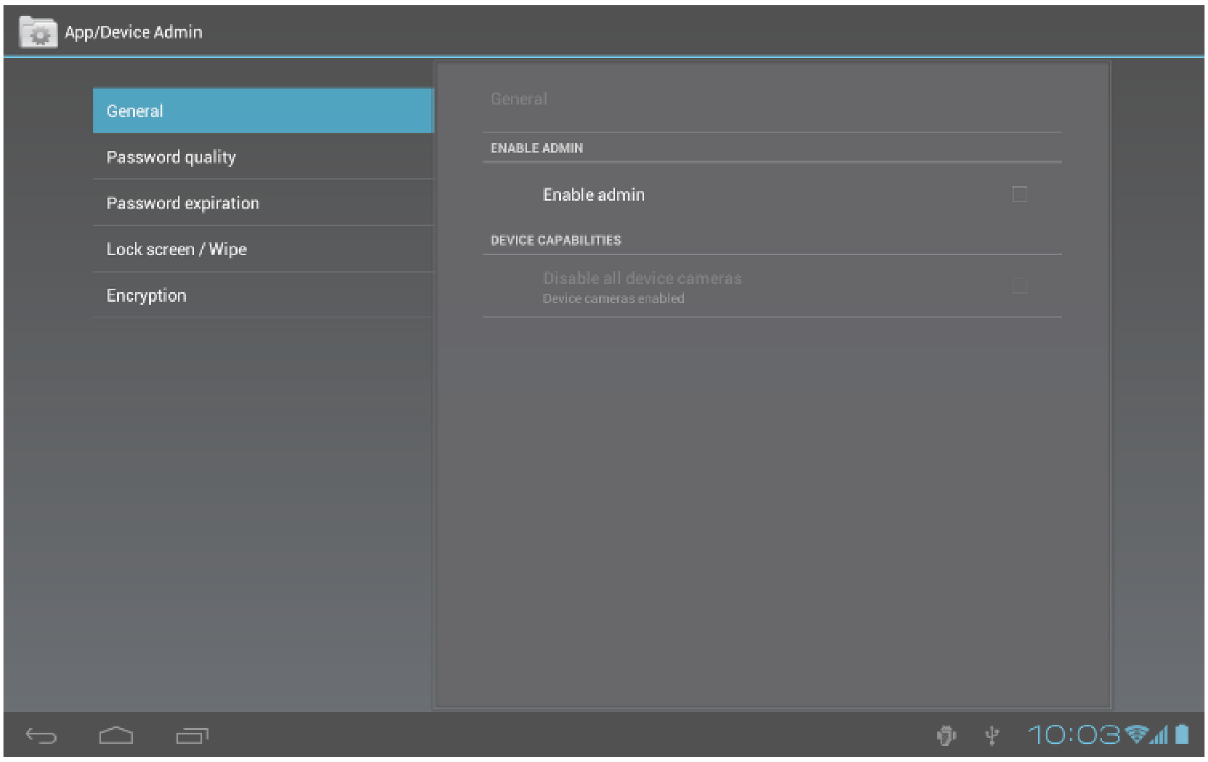


Figure 1. Screenshot of the Sample Application

Developing a Device Administration Application

System administrators can use the Device Administration API to write an application that enforces remote/local device security policy enforcement. This section summarizes the steps involved in creating a device administration application.

Creating the manifest

To use the Device Administration API, the application's manifest must include the following:

- A subclass of `DeviceAdminReceiver` that includes the following:
  - The `BIND_DEVICE_ADMIN` permission.
  - The ability to respond to the `ACTION_DEVICE_ADMIN_ENABLED` intent, expressed in the manifest as an intent filter.
- A declaration of security policies used in metadata.

Here is an excerpt from the Device Administration sample manifest:

```
<activity android:name=".app.DeviceAdminSample"
            android:label="@string/activity_sample_device_admin">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.SAMPLE_CODE" />
    </intent-filter>
</activity>
<receiver android:name=".app.DeviceAdminSample$DeviceAdminSampleReceiver"
        android:label="@string/sample_device_admin"
        android:description="@string/sample_device_admin_description"
        android:permission="android.permission.BIND_DEVICE_ADMIN">
    <meta-data android:name="android.app.device_admin"
            android:resource="@xml/device_admin_sample" />
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

Note that:

- The following attributes refer to string resources that for the sample application reside in `ApiDemos/res/values/strings.xml`. For more information about resources, see Application Resources.
    - `android:label="@string/activity_sample_device_admin"` refers to the user-readable label for the activity.
    - `android:label="@string/sample_device_admin"` refers to the user-readable label for the permission.
    - `android:description="@string/sample_device_admin_description"` refers to the user-readable description of the permission. A descripton is typically longer and more informative than a label.
- `android:permission="android.permission.BIND_DEVICE_ADMIN"` is a permission that a `DeviceAdminReceiver` subclass must have, to ensure that only the system can interact with the receiver (no application can be granted this permission). This prevents other applications from abusing your device admin app.
- `android.app.action.DEVICE_ADMIN_ENABLED` is the primary action that a `DeviceAdminReceiver` subclass must handle to be allowed to manage a device. This is set to the receiver when the user enables the device admin app. Your code typically handles this in `onEnabled()`. To be supported, the receiver must also require the `BIND_DEVICE_ADMIN` permission so that other applications cannot abuse it.
- When a user enables the device admin application, that gives the receiver permission to perform actions in response to the broadcast of particular system events. When suitable event arises, the application can impose a policy. For example, if the user attempts to set a new password that doesn't meet the policy requirements, the application can prompt the user to pick a different password that does meet the requirements.
- Avoid changing the receiver name after publishing your app. If the name in the manifest changes, device admin is disabled when users update the app. To learn more, see `<receiver>`.
- `android:resource="@xml/device_admin_sample"` declares the security policies used in metadata. The metadata provides additional information specific to the device administrator, as parsed by the `DeviceAdminInfo` class. Here are the contents of `device_admin_sample.xml`:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-policies>
    <limit-password />
    <watch-login />
    <reset-password />
    <force-lock />
    <wipe-data />
    <expire-password />
    <encrypted-storage />
    <disable-camera />
  </uses-policies>
</device-admin>
```

When designing your device administration application, you don't need to include all of the policies, just the ones that are relevant for your app.

For more discussion of the manifest file, see the Android Developers Guide.

Implementing the code

The Device Administration API includes the following classes:

`DeviceAdminReceiver`

Base class for implementing a device administration component. This class provides a convenience for interpreting the raw intent actions that are sent by the system. Your Device Administration application must include a `DeviceAdminReceiver` subclass.

`DevicePolicyManager`

A class for managing policies enforced on a device. Most clients of this class must have published a `DeviceAdminReceiver` that the user has currently enabled. The `DevicePolicyManager` manages policies for one or more `DeviceAdminReceiver` instances

`DeviceAdminInfo`

This class is used to specify metadata for a device administrator component.

These classes provide the foundation for a fully functional device administration application. The rest of this section describes how you use the `DeviceAdminReceiver` and `DevicePolicyManager` APIs to write a device admin application.

Subclassing DeviceAdminReceiver

To create a device admin application, you must subclass `DeviceAdminReceiver`. The `DeviceAdminReceiver` class consists of a series of callbacks that are triggered when particular events occur.

In its `DeviceAdminReceiver` subclass, the sample application simply displays a `Toast` notification in response to particular events. For example:

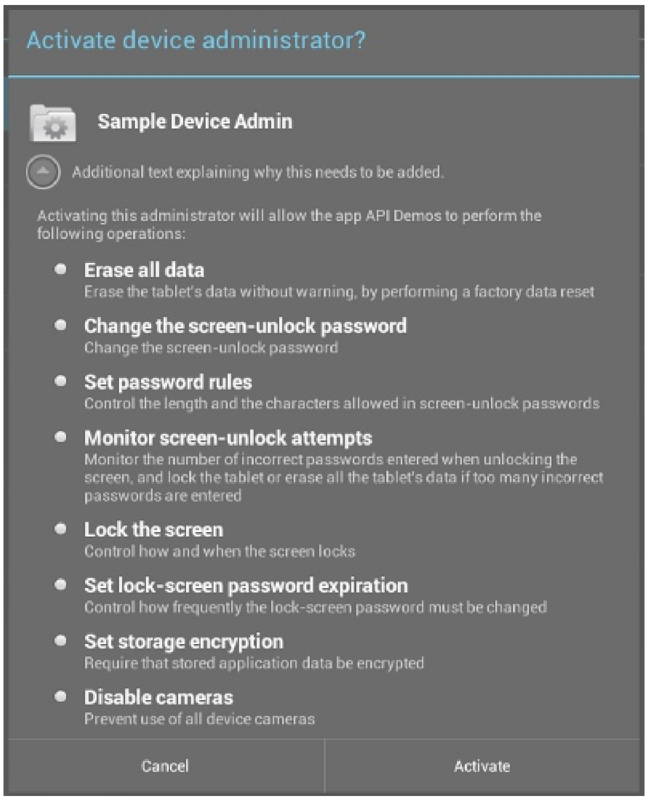
```
public class DeviceAdminSample extends DeviceAdminReceiver {

    void showToast(Context context, String msg) {
        String status = context.getString(R.string.admin_receiver_status, msg);
        Toast.makeText(context, status, Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onEnabled(Context context, Intent intent) {
        showToast(context, context.getString(R.string.admin_receiver_status_enabled));
    }

    @Override
    public CharSequence onDisableRequested(Context context, Intent intent) {
        return context.getString(R.string.admin_receiver_status_disable_warning);
    }

    @Override
    public void onDisabled(Context context, Intent intent) {
        showToast(context, context.getString(R.string.admin_receiver_status_disabled));
    }

    @Override
    public void onPasswordChanged(Context context, Intent intent) {
        showToast(context, context.getString(R.string.admin_receiver_status_pw_changed));
    }
...
}
```

Enabling the application

One of the major events a device admin application has to handle is the user enabling the application. The user must explicitly enable the application for the policies to be enforced. If the user chooses not to enable the application it will still be present on the device, but its policies will not be enforced, and the user will not get any of the application's benefits.

The process of enabling the application begins when the user performs an action that triggers the `ACTION_ADD_DEVICE_ADMIN` intent. In the sample application, this happens when the user clicks the **Enable Admin** checkbox.

When the user clicks the **Enable Admin** checkbox, the display changes to prompt the user to activate the device admin application, as shown in figure 2.


Activate device administrator?

Sample Device Admin

Additional text explaining why this needs to be added.

Activating this administrator will allow the app API Demos to perform the following operations:

- **Erase all data**
  Erase the tablet's data without warning, by performing a factory data reset
- **Change the screen-unlock password**
  Change the screen-unlock password
- **Set password rules**
  Control the length and the characters allowed in screen-unlock passwords
- **Monitor screen-unlock attempts**
  Monitor the number of incorrect passwords entered when unlocking the screen, and lock the tablet or erase all the tablet's data if too many incorrect passwords are entered
- **Lock the screen**
  Control how and when the screen locks
- **Set lock-screen password expiration**
  Control how frequently the lock-screen password must be changed
- **Set storage encryption**
  Require that stored application data be encrypted
- **Disable cameras**
  Prevent use of all device cameras

Figure 2. Sample Application: Activating the Application

Below is the code that gets executed when the user clicks the **Enable Admin** checkbox. This has the effect of triggering the `onPreferenceChange()` callback. This callback is invoked when the value of this `Preference` has been changed by the user and is about to be set and/or persisted. If the user is enabling the application, the display changes to prompt the user to activate the device admin application, as shown in figure 2. Otherwise, the device admin application is disabled.

```
@Override
        public boolean onPreferenceChange(Preference preference, Object newValue) {
            if (super.onPreferenceChange(preference, newValue)) {
                return true;
            }
            boolean value = (Boolean) newValue;
            if (preference == mEnableCheckbox) {
                if (value != mAdminActive) {
                    if (value) {
                        // Launch the activity to have the user enable our admin.
                        Intent intent = new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
                        intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, mDeviceAdminSample);
                        intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                                mActivity.getString(R.string.add_admin_extra_app_text));
                        startActivityForResult(intent, REQUEST_CODE_ENABLE_ADMIN);
                        // return false - don't update checkbox until we're really active
                        return false;
                    } else {
                        mDPM.removeActiveAdmin(mDeviceAdminSample);
                        enableDeviceCapabilitiesArea(false);
                        mAdminActive = false;
                    }
                }
            } else if (preference == mDisableCameraCheckbox) {
                mDPM.setCameraDisabled(mDeviceAdminSample, value);
                ...
            }
            return true;
        }
```

The line `intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, mDeviceAdminSample)` states that `mDeviceAdminSample` (which is a `DeviceAdminReceiver` component) is the target policy. This line invokes the user interface shown in figure 2, which guides users through adding the device administrator to the system (or allows them to reject it).

When the application needs to perform an operation that is contingent on the device admin application being enabled, it confirms that the application is active. To do this it uses the `DevicePolicyManager` method `isAdminActive()`. Notice that the `DevicePolicyManager` method `isAdminActive()` takes a `DeviceAdminReceiver` component as its argument:

```
DevicePolicyManager mDPM;
...
private boolean isActiveAdmin() {
    return mDPM.isAdminActive(mDeviceAdminSample);
}
```

Managing policies

`DevicePolicyManager` is a public class for managing policies enforced on a device. `DevicePolicyManager` manages policies for one or more `DeviceAdminReceiver` instances.

You get a handle to the `DevicePolicyManager` as follows:

```
DevicePolicyManager mDPM =
    (DevicePolicyManager)getSystemService(Context.DEVICE_POLICY_SERVICE);
```

This section describes how to use `DevicePolicyManager` to perform administrative tasks:

- Set password policies
- Set device lock
- Perform data wipe

Set password policies

`DevicePolicyManager` includes APIs for setting and enforcing the device password policy. In the Device Administration API, the password only applies to screen lock. This section describes common password-related tasks.

Set a password for the device

This code displays a user interface prompting the user to set a password:

```
Intent intent = new Intent(DevicePolicyManager.ACTION_SET_NEW_PASSWORD);
startActivity(intent);
```

Set the password quality

The password quality can be one of the following `DevicePolicyManager` constants:

`PASSWORD_QUALITY_ALPHABETIC`

The user must enter a password containing at least alphabetic (or other symbol) characters.

`PASSWORD_QUALITY_ALPHANUMERIC`

The user must enter a password containing at least *both* numeric *and* alphabetic (or other symbol) characters.

`PASSWORD_QUALITY_NUMERIC`

The user must enter a password containing at least numeric characters.

`PASSWORD_QUALITY_COMPLEX`

The user must have entered a password containing at least a letter, a numerical digit and a special symbol.

`PASSWORD_QUALITY_SOMETHING`

The policy requires some kind of password, but doesn't care what it is.

`PASSWORD_QUALITY_UNSPECIFIED`

The policy has no requirements for the password.

For example, this is how you would set the password policy to require an alphanumeric password:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
...
mDPM.setPasswordQuality(mDeviceAdminSample, DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC);
```

Set password content requirements

Beginning with Android 3.0, the `DevicePolicyManager` class includes methods that let you fine-tune the contents of the password. For example, you could set a policy that states that passwords must contain at least *n* uppercase letters. Here are the methods for fine-tuning a password's contents:

- `setPasswordMinimumLetters()`
- `setPasswordMinimumLowerCase()`
- `setPasswordMinimumUpperCase()`
- `setPasswordMinimumNonLetter()`
- `setPasswordMinimumNumeric()`
- `setPasswordMinimumSymbols()`

For example, this snippet states that the password must have at least 2 uppercase letters:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
int pwMinUppercase = 2;
...
mDPM.setPasswordMinimumUpperCase(mDeviceAdminSample, pwMinUppercase);
```

Set the minimum password length

You can specify that a password must be at least the specified minimum length. For example:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
int pwLength;
...
mDPM.setPasswordMinimumLength(mDeviceAdminSample, pwLength);
```

Set maximum failed password attempts

You can set the maximum number of allowed failed password attempts before the device is wiped (that is, reset to factory settings). For example:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
int maxFailedPw;
 ...
mDPM.setMaximumFailedPasswordsForWipe(mDeviceAdminSample, maxFailedPw);
```

Set password expiration timeout

Beginning with Android 3.0, you can use the `setPasswordExpirationTimeout()` method to set when a password will expire, expressed as a delta in milliseconds from when a device admin sets the expiration timeout. For example:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
long pwExpiration;
...
mDPM.setPasswordExpirationTimeout(mDeviceAdminSample, pwExpiration);
```

Restrict password based on history

Beginning with Android 3.0, you can use the `setPasswordHistoryLength()` method to limit users' ability to reuse old passwords. This method takes a *length* parameter, which specifies how many old passwords are stored. When this policy is active, users cannot enter a new password that matches the last *n* passwords. This prevents users from using the same password over and over. This policy is typically used in conjunction with `setPasswordExpirationTimeout()`, which forces users to update their passwords after a specified amount of time has elapsed.

For example, this snippet prohibits users from reusing any of their last 5 passwords:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
int pwHistoryLength = 5;
...
mDPM.setPasswordHistoryLength(mDeviceAdminSample, pwHistoryLength);
```

Set device lock

You can set the maximum period of user inactivity that can occur before the device locks. For example:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
...
long timeMs = 1000L*Long.parseLong(mTimeout.getText().toString());
mDPM.setMaximumTimeToLock(mDeviceAdminSample, timeMs);
```

You can also programmatically tell the device to lock immediately:

```
DevicePolicyManager mDPM;
mDPM.lockNow();
```

Perform data wipe

You can use the `DevicePolicyManager` method `wipeData()` to reset the device to factory settings. This is useful if the device is lost or stolen. Often the decision to wipe the device is the result of certain conditions being met. For example, you can use `setMaximumFailedPasswordsForWipe()` to state that a device should be wiped after a specific number of failed password attempts.

You wipe data as follows:

```
DevicePolicyManager mDPM;
mDPM.wipeData(0);
```

The `wipeData()` method takes as its parameter a bit mask of additional options. Currently the value must be 0.

Disable camera

Beginning with Android 4.0, you can disable the camera. Note that this doesn't have to be a permanent disabling. The camera can be enabled/disabled dynamically based on context, time, and so on.

You control whether the camera is disabled by using the `setCameraDisabled()` method. For example, this snippet sets the camera to be enabled or disabled based on a checkbox setting:

```
private CheckBoxPreference mDisableCameraCheckbox;
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
...
mDPM.setCameraDisabled(mDeviceAdminSample, mDisableCameraCheckbox.isChecked());
```

Storage encryption

Beginning with Android 3.0, you can use the `setStorageEncryption()` method to set a policy requiring encryption of the storage area, where supported.

For example:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
...
mDPM.setStorageEncryption(mDeviceAdminSample, true);
```

See the Device Administration API sample for a complete example of how to enable storage encryption.