The Android secure storage is based on JCA (Java cryptography architecture )

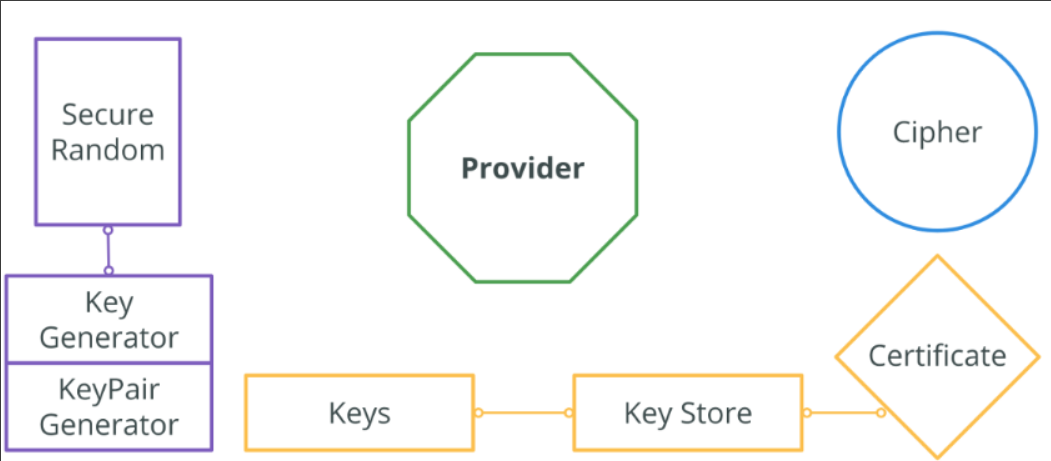
If the device is rooted , the information stored in the shared preference is no more secure.

In order to make full proof from App perspective , irrespective of the device on which app is installed , we can use Android keyStore  with encryption ,decryption techniques .

Eg: Need of secure storage :

This course will guide you on theory and practical usage of various components involved in this process.

Java Cryptography architecture



**KeyGenerator**

* provides the public API for generating symmetric cryptographic keys.

**KeyPairGenerator**

* an engine class which is capable of generating a private key and its related public key utilizing the algorithm it was initialized with.

**SecretKey**

* a secret (symmetric) key. The purpose of this interface is to group (and provide type safety for) all secret key interfaces (e.g.,*SecretKeySpec*).

**PrivateKey**

* a private (asymmetric) key. The purpose of this interface is to group (and provide type safety for) all private key interfaces(e.g., *RSAPrivateKey*).

**PublicKey**

* a public key. This interface contains no methods or constants. It merely serves to group (and provide type safety for) all public key interfaces(e.g.,*RSAPublicKey*).

**KeyPair**

* this class is a simple holder for a key pair (a public key and a private key). It does not enforce any security, and, when initialized, should be treated like a PrivateKey.

**SecureRandom**

* generates cryptographically secure pseudo-random numbers. We will not use it directly in this series, but it is widely used inside of *KeyGenerator, KeyPairGenerator* components and *Keys* implementations.

**KeyStore**

* database with a well secured mechanism of data protection, that is used to save, get and remove keys. Requires entrance password and passwords for each of the keys. In other words it is protected file that you need to create, read and update (with provided API).

**Certificate**

* certificate used to validate and save asymmetric keys.

**Cipher**

* provides access to implementations of cryptographic ciphers for encryption, decryption, wrapping, unwrapping and signing.

**Provider**

* defines a set of extensible implementations, independent API’s. *Providers* are the groups of different Algorithms or their customizations. There are 3d party providers, such as [Bouncy Castle](https://www.bouncycastle.org/) and [Spongy Castle](https://rtyley.github.io/spongycastle/) (android version of Bouncy Castle), as well as providers available out of box, such as cut-down version of Bouncy Castle.

**Source code of sample App:**

The App don't concentrate on UI , it is build just for realizing the concepts .

Reference:

<https://developer.android.com/training/articles/keystore#java>

<https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html>

<https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html#KeyManager>

<https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyStore>

<https://proandroiddev.com/secure-data-in-android-encrypting-large-data-dda256a55b36>

# [0.1 Android Keystore Introduction](https://confluence.rakuten-it.com/confluence/display/~amit.nadiger/0.1+Android+Keystore+Introduction)

## **What is Keystore:**

Key store is kind of locker where where you protect Keys . How locker is protected  => In Java case by password.

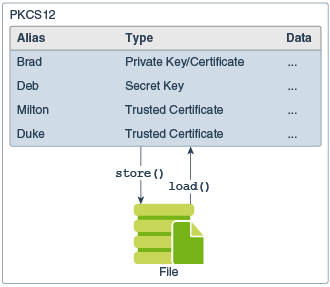
A Java KeyStore is represented by the KeyStore(java.security.KeyStore) class.

A KeyStore can be written to disk and read again. The KeyStore as a whole can be protected with a password, and each key entry in the KeyStore can be protected with its own password.

This makes the KeyStore class a useful mechanism to handle encryption keys securely.

Example for Java key store : **xxx.keystore**  file which is used to  sign the android APK . This keystore contains the keys used to  sign the APK .

The KeyStore class is an [engine class](https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#Engine) that supplies well-defined interfaces to access and modify the information in a keystore.



This class represents an in-memory collection of keys and certificates

## **AndroidKeystore :**

* It is Android Framework API and component used by apps to access Keystore functionality. It is implemented as an extension to the standard Java Cryptography Architecture APIs, and consists of Java code that runs in the app's own process space.
* AndroidKeystore fulfills app requests for Keystore behavior by forwarding them to the keystore daemon.

## **What is unique about AndroidKeystore**

* Lets you store cryptographic keys in a container to make it more difficult to extract from the device.
* Once keys are in the key store, they can be used for cryptographic operations with the key material remaining non-exportable.
* **No KeyStore passwords is required.  <= This is the main difference between java keystore(password is must) and Android kestore(No Password)**
* Key material never enters the application process
* Key material may be bound to the secure hardware ([Trust Zone](https://en.wikipedia.org/wiki/Trusted_execution_environment))
* Asymmetric keys are available from 18 +
* Symmetric keys are available from 23 +

## **What is Trust Zone:**

Key material may be bound to the secure hardware (e.g., Trusted Execution Environment (TEE), Secure Element (SE)) of the Android device. When this feature is enabled for a key, its key material is never exposed outside of secure hardware.

If the Android OS is compromised or an attacker can read the device's internal storage, the attacker may be able to use any app's Android Keystore keys on the Android device, but not extract them from the device.

* A **trusted execution environment** (**TEE**) is a secure area of a main processor. It guarantees code and data loaded inside to be protected with respect to confidentiality and integrity.
* If device manufacture supports Trusted Execution Environment(TEE), your keys will be saved there (the most secure option);
* If device manufacture doesn’t support TEE, keys will be stored in emulated software environment, provided by the system.

## **What are Keys :**

They are cryptographic keys , used to encrypt and decrypt data .

Keys can be symmetric(same key i.e secret Key ) or asymmetric key (private and public keys).

## **keystore daemon:**

* It is Android system daemon that provides access to all Keystore functionality via a [Binder API](https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/security/IKeystoreService.aidl).
* It's responsible for storing "key blobs", which contain the actual secret key material, encrypted so Keystore can store it but not use it or reveal it.

## **keymasterd:**

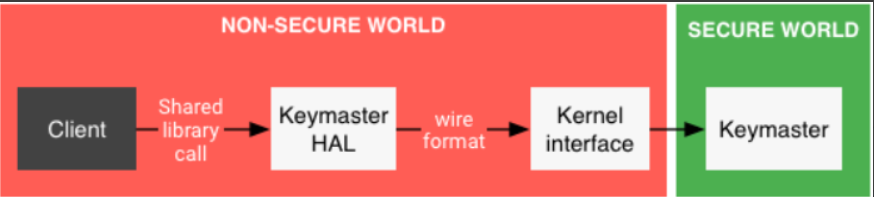
* is a HIDL server that provides access to the Keymaster TA.
* HIDL: HAL interface definition language .  is an interface description language (IDL) to specify the interface between a HAL and its users. More broadly, HIDL is a system for communicating between codebases that may be compiled independently.

## **Architecture:**

The Android Keystore API and the underlying Keymaster HAL provides a basic but adequate set of cryptographic primitives to allow the implementation of protocols using access-controlled, hardware-backed keys.

The Keymaster HAL is an OEM-provided, dynamically-loadable library used by the Keystore service to provide hardware-backed cryptographic services. To keep things secure, HAL implementations don't perform any sensitive operations in user space, or even in kernel space.

Sensitive operations are delegated to a secure processor reached through some kernel interface. The resulting architecture looks like this:



The objective is to store  the store sensitive information , so that no one but just that app on that device can access the sensitive Info.

**This is achieved as below .**

1. Create & load  the Keystore .
2. Generate the keys (Symmetric or Asymmetric.)
3. Encrypt the sensitive information.
   1. can uses symmetric  or combination of asymmetric and symmetric algo
4. Keys used will be stored in keystore
5. Encrypted information will be stored in shared preference or other storage.
6. When need to access the information , get the encrypted info from storage and decrypt it using keys stored in key store.
   1. Here keys are stored in keystore  , so no one but app can retrieve these keys.
   2. Since Keys are safe , our sensitive info is also safe.

## **Components involved in this process**

1. Keystore
2. Key Generations
   1. Key generator  (Symmetric )
   2. Key Pair generator (Asymmetric)
      1. Master key which will be symmetric key
      2. This master key will be encrypted by asymmetric keys using public key which is stroed in key-store , this is called wrapping
      3. encrypted master key will be stored in shared preference.
      4. During decryption , reverse of  steps ii,iii will be done. (This unwrapping)
3. Cipher
4. Initialization Vector (IV)
5. Storage media

**1. Keystore**

### ****Creating a****KeyStore****Object****

* KeyStore objects are obtained by using one of the KeyStore [getInstance(type) static factory methods](https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderImplReq).
* Here Type can be many types ; Like  jks, pkcs12, "AndroidKeyStore ,etc..All these types are provided by different providers.
* For Android we use :**"AndroidKeyStore" provider.**
* Implementation looks like KeyStore.getInstance(ANDROID\_KEY\_STORE); // keyStore.provider.name= AndroidKeyStore and keyStore.type = AndroidKeyStore
* Default Keystore  type in Android device is BKS (BouncyCastle) <= It should not be used.
  + KeyStore.getInstance(KeyStore.getDefaultType()); <+= here KeyStore.getDefaultType() return BKS.

### Loading a Particular Keystore into Memory

* Before a KeyStore object can be used, the actual keystore data must be loaded into memory via the load method:  => mkeyStore.load(null);
* To create an empty keystore, you pass null as the InputStream argument to the load method.
* =>  Implementation looks like mkeyStore.load(null);

### Implementation :

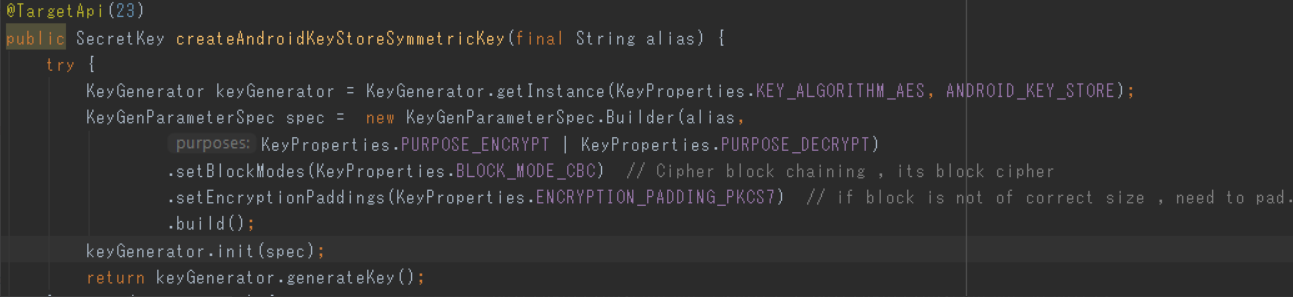


## **2. Key Generation**

* In order to encrypt and decrypt the sensitive info such as Token , user info , we need Keys.  Keye are based on type of algorithm we (Symmetric - Key generator or Asymmetric - Key pair generator)

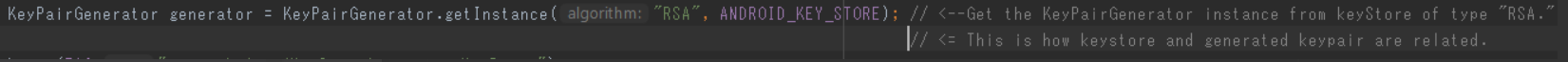
### For API level 23+

**2. 1 KeyGenerator:** (Symmetric  cryptography : Same key used for encryption and Decryption) =>  available from 23+ [No restriction on size of data to be encrypted , Divides big chunk of data as blocks , and encrypt  individual blocks]

* KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY\_ALGORITHM\_AES, **ANDROID\_KEY\_STORE**)   <--Get the KeyGenerator instance from keyStore of type AES .  <= This is how keystore and generated keys are related.
* Need  to have spec to generate the keys. (Alias (), purpose , blocks , type of padding for  irregular size blocks  )  
  **Implementation:**  
  

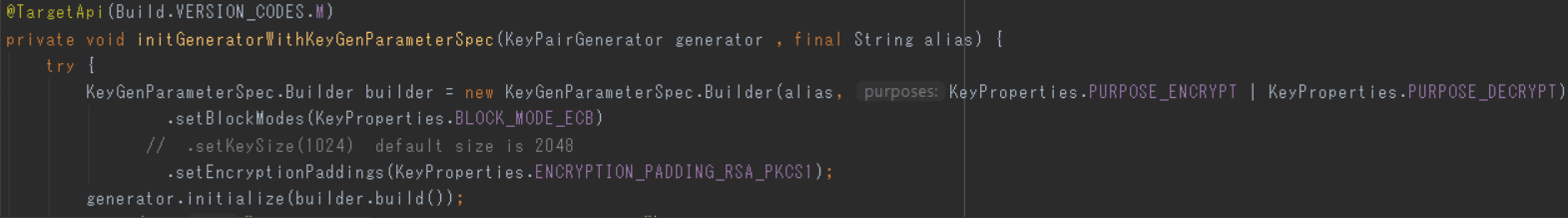
**Note** : If the device is 23+ , one should use Android Symmetric key cryptography .

**2.2** **KeyPairGenerator :**(Asymmetric  cryptography : publicKey-Encryption and privateKey-Decryption) =>  available from 18+ [restriction on size of data to be encrypted ]

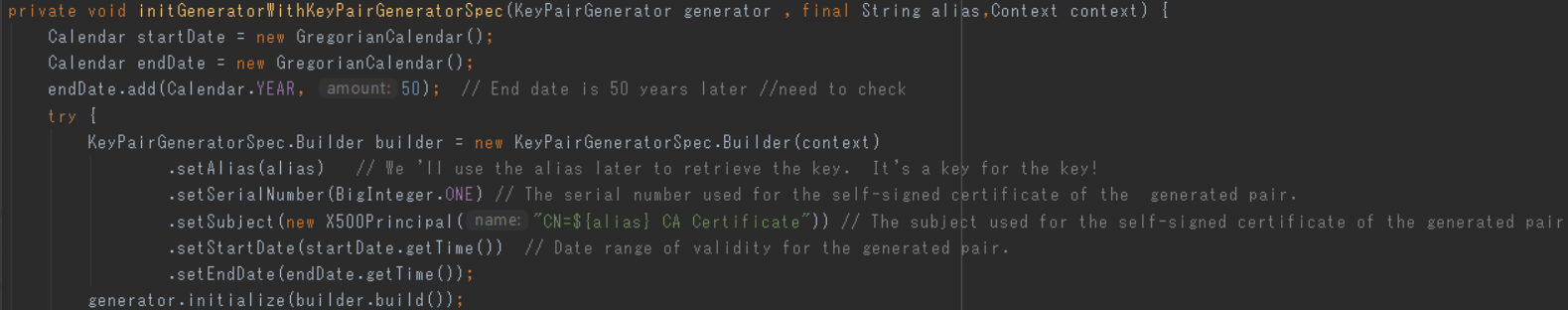


* There are 2 ways to generate the key-pair based on spec :

1. With **KeyGenParameterSpec** : Should be used with device >= APL 23

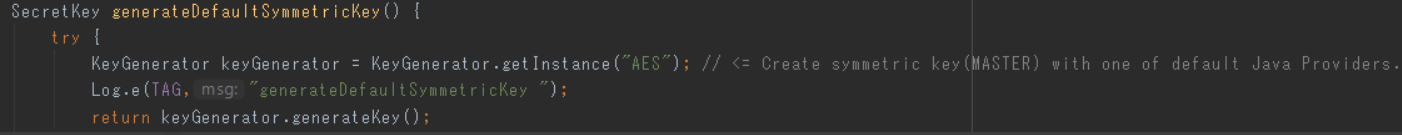


1. With **KeyPairGeneratorSpec**: Should be used with device >=  API18 and < API 23

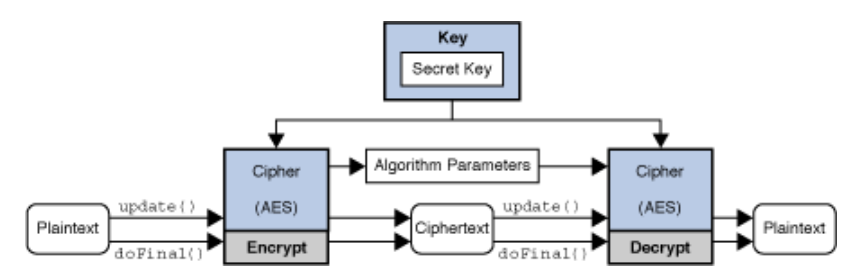


**3. Master Key generation :**

* This is independent of keyStore .(Used in device with api >=18 && <23  for big data )
* Create symmetric key(MASTER) with one of default Java Providers.  The most common default Java provider in android is the cut version of BC provider created by the popular third party Java cryptographic library provider — [Bouncy Castle](https://bouncycastle.org/specifications.html.).
* Encrypt / decrypt message with it.
* Then encrypt this key raw data with RSA public key and save it in shared preference or somewhere.
* On decryption, get encrypted raw key data,  decrypt it with RSA private key and use it for message decryption.



# [0.3 Encryption ,Decryption](https://confluence.rakuten-it.com/confluence/display/~amit.nadiger/0.3+Encryption+%2CDecryption)



Need cipher class for cryptographic functionality : Encryption and decryption.

Cipher is independent module , where it encrypts and decrypt the data or info via Key.

## **Types of Encryption/Decryption:**

There are 3 types of Encryption/ Decryption

1. Symmetric(secret key)        <=  same secret key to both encrypt and decrypt the data.
2. Asymmetric(public key cryptography) <= uses a public(Encryption)/private(Decryption) key pair to encrypt data(Usually case ). In below case Reverse is used (Public: Decrypt , Private: Encrypt). Asymmetric is not suitable for large data , Please see Hybrid for this.
   1. In case of Digital signature authorization => the authority uses their private key to encrypt the contents of the certificate, and this cipher text is attached to the certificate as its digital signature.
   2. Anyone can decrypt this signature using the authority’s public key, and verify that it results in the expected decrypted value.
   3. Only the authority can encrypt content using the private key, and so only the authority can actually create a valid signature in the first place.
3. Hybrid (Symmetric + Asymmetric )  <= Symmetric (Secret) Key used to encrypt, decrypts the actual info , This secret key it self encrypted using Asymmetric (public/private).

## **Overall flow management of Keystore , encryption ,Decryption , Cipher :**

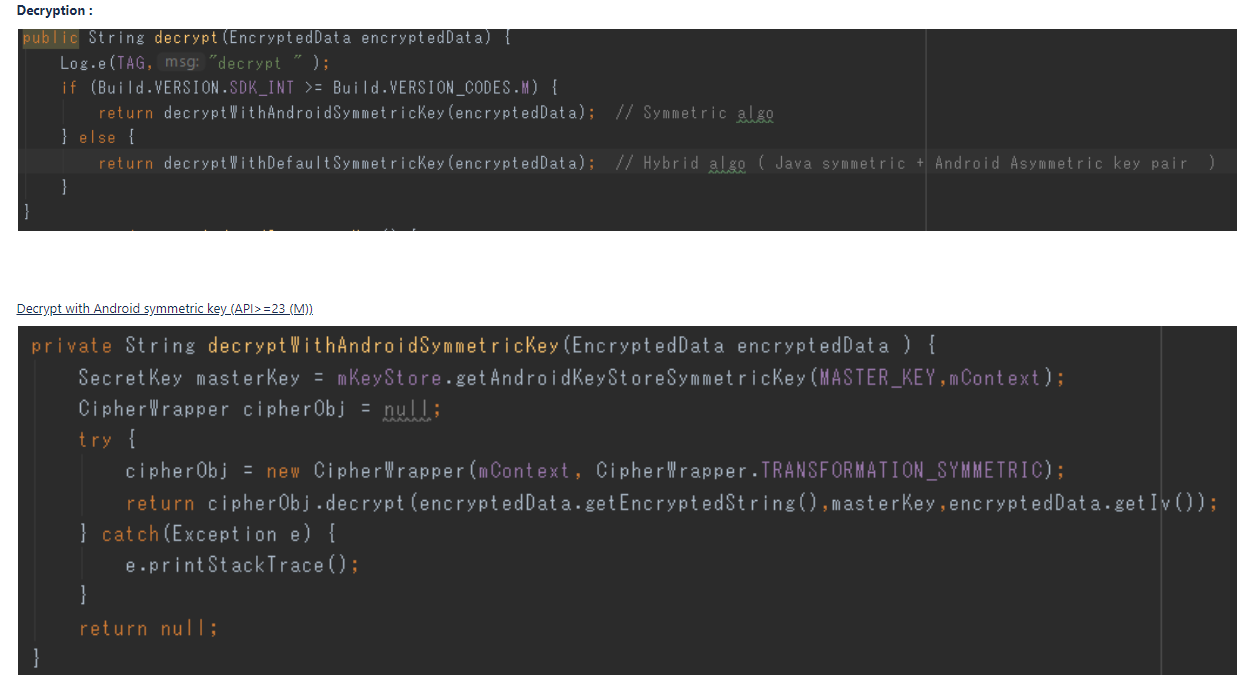
1. Create a class called**"**CryptoService.java" which abstracts  following component  . So  it acts as block box containing "Keystore , cipher (encryption/Decryption )".
2. It provides interfaces for below functionality:
   1. Encryption
   2. Decryption
   3. Storing the encrypted Master Key in shared preference (in device APIL<23)
3. **"**CryptoService". class handles below decisions based on android version(API level ) and type of encryption ,Decryption
   1. If API Level >=23
      1. Create Master key (Android symmetric key ) if API level >=23
   2. If API Level <23 and >=18
      1. Create Master key (default  java provided symmetric key) if API level <23 and >=18.
      2. Create the Keppair(Private + Public) if   if API level <23 and >=18. Using "KeyPairGeneratorSpec "
         1. If API level <23 and <= 18 => we must use Using "KeyPairGeneratorSpec " for generating the key pair
         2. If API level >23 we must use "KeyGenParameterSpec" as "KeyPairGeneratorSpec " is deprecated <= does not happen as we use symmetric key if API >23. If we want use specifically asymmetric crypto in device API>23 , then use "KeyGenParameterSpec".
      3. Encrypt the Master Key generated in i step with public key (This is called wrap)  <= when need to store in shared preference or secure transfer of hardware-based keys.
      4. Decrypt the Master Key encrypted in iii step with privatekey (This is called unwrap) <= when need to encrypt the sensitive information.

**Create Master Key :**











# [0.4 Cipher](https://confluence.rakuten-it.com/confluence/display/~amit.nadiger/0.4+Cipher)

## **Type of  ciphers:**

**Block:**

* **P**rocess entire blocks at a time, usually many bytes in length.
* If there is not enough data to make a complete input block, the data must be padded to match cipher's block size.
* The padded bytes are then stripped off during the decryption phase.
* EX : "PKCS5PADDING, ,PKCS7Padding,PKCS1Padding ,etc

**Stream:**

* Process incoming data one small unit (typically a byte or even a bit) at a time.
* This allows for ciphers to process an arbitrary amount of data without padding.

## **Modes of Operation :**

* When encrypting using a simple block cipher, two identical blocks of plaintext will always produce an identical block of cipher text. Cryptanalysts trying to break the ciphertext will have an easier job if they note blocks of repeating text.
* In order to add more complexity to the text, feedback modes use the previous block of output to alter the input blocks before applying the encryption algorithm.
* The first block will need an initial value, and this value is called the initialization vector (IV)
* IV can be random and need not be secret .
* Modes EX:
  + CBC (Cipher Block Chaining),  <=  each cipher data block depends on all plain data blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block.
  + CFB (Cipher Feedback Mode),
  + OFB (Output Feedback Mode).
  + ECB (Electronic Codebook Mode)  <= ECB ciphertexts are the same if they use the same plaintext/key , Dont use for encryption.

***Note:*** ECB mode is the easiest block cipher mode to use and is the default in the JDK/JRE. ECB works well for single blocks of data, but absolutely should not be used for multiple data blocks

## **1. Creating the Cipher Object:**

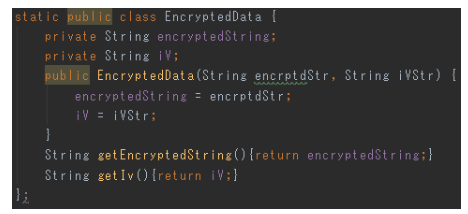
### mCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");  <=  asymmetric cipher

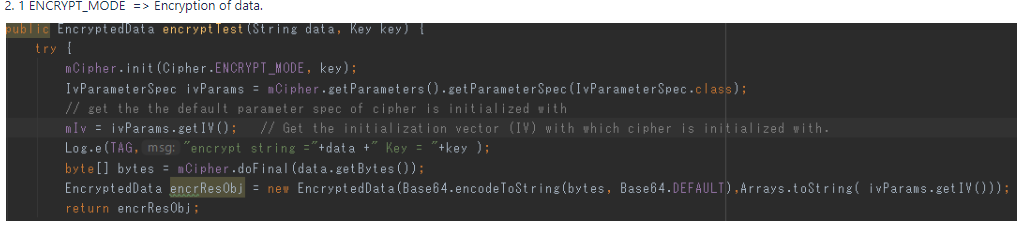
### mCipher = Cipher.getInstance("AES/CBC/PKCS7Padding);    <=  symmetric cipher

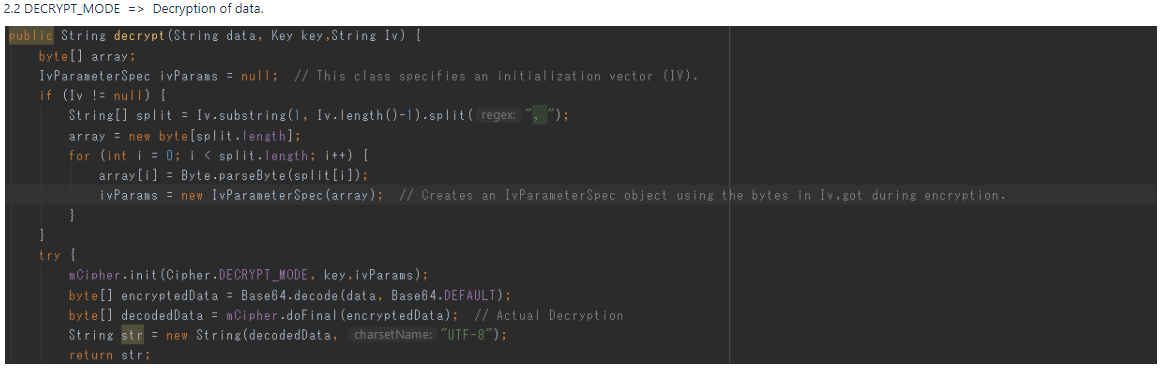
## **2. Initializing a Cipher Object**

A Cipher object obtained via getInstance must be initialized for one of four modes, which are defined as final integer constants in the Cipher class.

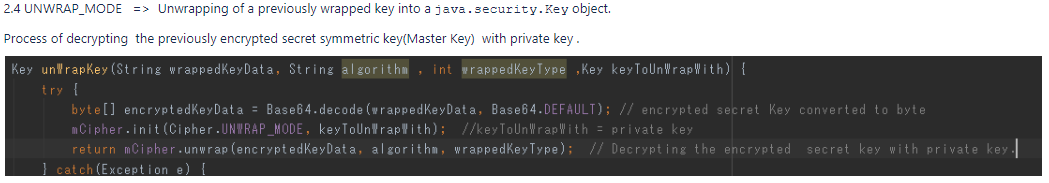
EncryptedData  : The output of  encryption and input to Decryption.











# [0.5 SecureStorageManager](https://confluence.rakuten-it.com/confluence/display/~amit.nadiger/0.5+SecureStorageManager)

### SecureStorageManager is wrapper class for shared preference where data is 1st encrypted and then stored .

## **Mechanism or How it works :**

**Storing the data :**

1. Get the plane or clear data in  Key-Value pair , same as shared preference from the client.
2. Encrypt the data or Value with "**CryptoService"** instance  which internally uses Keys from keystore.
3. Store the encrypted data as Key(clear Text)-Value(Encrypted)

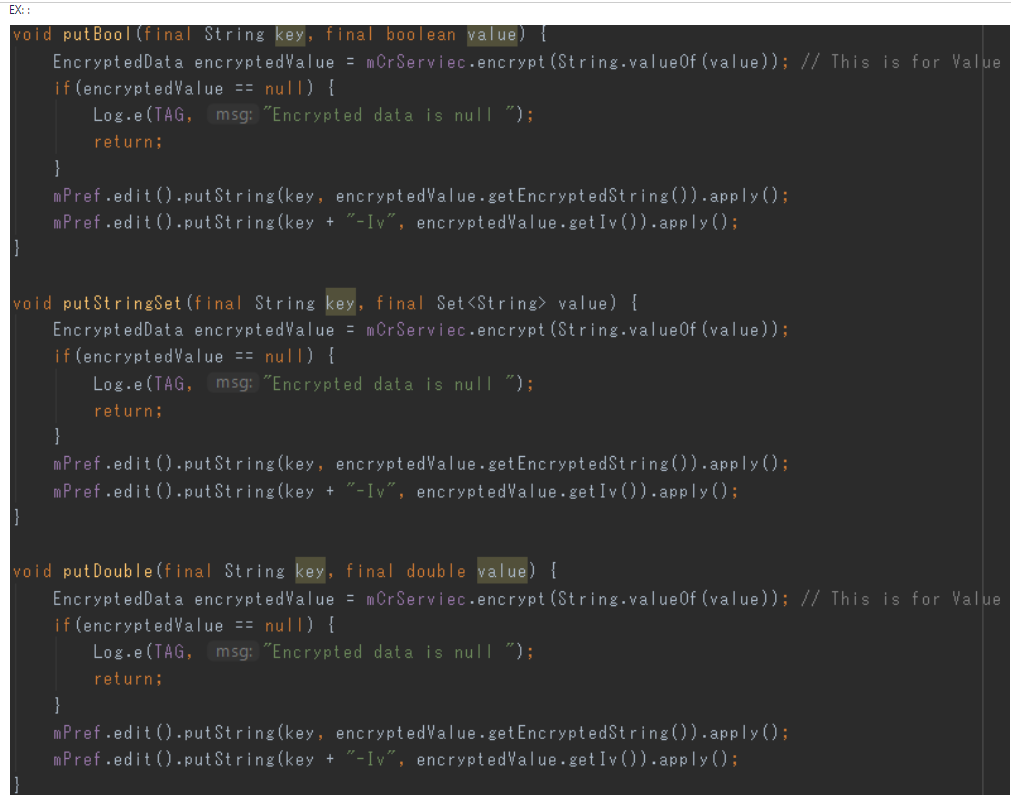
**Retrieving the data :**

1. Get the encrypted  data from shared preference based on the  Key used while storing .
2. De-crypt the data or Value with "**CryptoService"** instance  which internally uses Keys from keystore.
3. Provide the clear or plane data to client

## **Functionalities :**

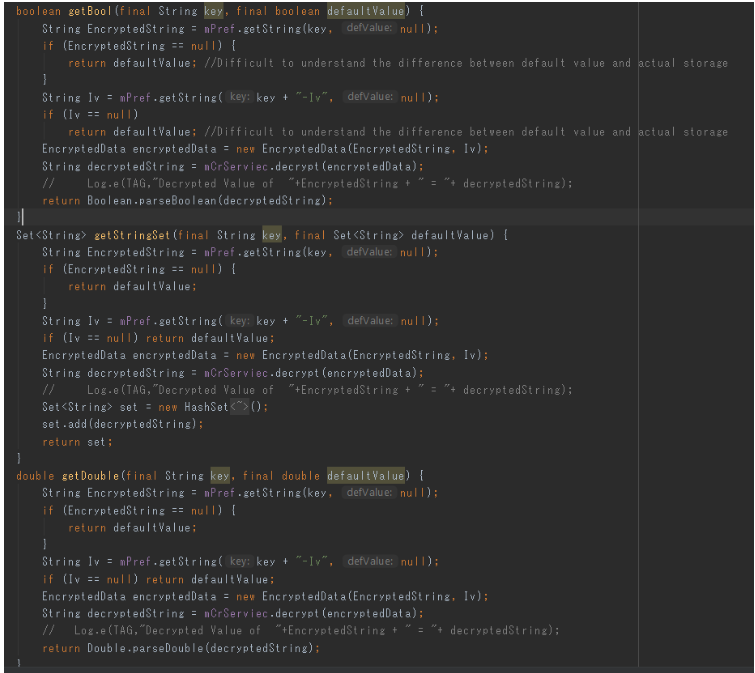
**Storage Methods:**

1. void putInt(final String key, final int value)
2. void putLong(final String key, final long value)
3. void putString(final String key, final String value)
4. void putFloat(final String key, final float value)
5. void putBool(final String key, final boolean value)
6. void putStringSet(final String key, final Set<String> value)
7. void putDouble(final String key, final double value)



**Retrieve Methods :**

1. int getInt(final String key, final int defaultValue)
2. long getLong(final String key, final long defaultValue)
3. String getString(final String key, final String defaultValue)
4. float getFloat(final String key, final float defaultValue)
5. boolean getBool(final String key, final boolean defaultValue)
6. Set<String> getStringSet(final String key, final Set<String> defaultValue)
7. doublegetDouble(final String key, final double defaultValue)



**Additional Methods :**

1. boolean contains(final String key)
2. void removeKey(final String key)
3. void clearStorage(final String key)

## **How does Plane data is stored in shared preference (rooted device):**



## **How does Encrypted data is stored in shared preferenc(rooted device):**

