**Java KeyGenerator**

The *Java KeyGenerator* class (javax.crypto.KeyGenerator) is used to generate symmetric encryption keys. A symmetric encryption key is a key that is used for both encryption and decryption of data, by a symmetric encryption algorithm. In this Java KeyGenerator tutorial I will show you how to generate symmetric encryption keys.

**Creating a KeyGenerator Instance**

Before you can use the Java KeyGenerator class you must create a KeyGenerator instance. You create a KeyGenerator instance by calling the static method getInstance() passing as parameter the name of the encryption algorithm to create a key for. Here is an example of creating a Java KeyGenerator instance:

KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");

This example creates a KeyGenerator instance which can generate keys for the AES encryption algorithm.

**Initializing the KeyGenerator**

After creating the KeyGenerator instance you must initialize it. Initializing a KeyGenerator instance is done by calling its init() method. Here is an example of initializing a KeyGenerator instance:

SecureRandom secureRandom = new SecureRandom();

int keyBitSize = 256;

keyGenerator.init(keyBitSize, secureRandom);

The KeyGenerator init() method takes two parameters: The bit size of the keys to generate, and a SecureRandom that is used during key generation.

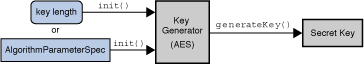
**Generating a Key**

Once the Java KeyGenerator instance is initialized you can use it to generate keys. Generating a key is done by calling the KeyGenerator generateKey() method. Here is an example of generating a symmetric key:

SecretKey secretKey = keyGenerator.generateKey();

## The KeyGenerator Class

A key generator is used to generate secret keys for symmetric algorithms.



[Description of Figure 13: The KeyGenerator Class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "keygenerator)

### Creating a KeyGenerator

KeyGenerator objects are obtained by using one of the KeyGenerator [getInstance() static factory methods](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderImplReq).

### Initializing a KeyGenerator Object

A key generator for a particular symmetric-key algorithm creates a symmetric key that can be used with that algorithm. It also associates algorithm-specific parameters (if any) with the generated key.

There are two ways to generate a key: in an algorithm-independent manner, and in an algorithm-specific manner. The only difference between the two is the initialization of the object:

* **Algorithm-Independent Initialization**

All key generators share the concepts of a *keysize* and a *source of randomness*. There is an init method that takes these two universally shared types of arguments. There is also one that takes just a keysize argument, and uses a system-provided source of randomness, and one that takes just a source of randomness:

public void init(SecureRandom random);

public void init(int keysize);

public void init(int keysize, SecureRandom random);

Since no other parameters are specified when you call the above algorithm-independent init methods, it is up to the provider what to do about the algorithm-specific parameters (if any) to be associated with the generated key.

* **Algorithm-Specific Initialization**

For situations where a set of algorithm-specific parameters already exists, there are two init methods that have an AlgorithmParameterSpec argument. One also has a SecureRandom argument, while the source of randomness is system-provided for the other:

public void init(AlgorithmParameterSpec params);

public void init(AlgorithmParameterSpec params, SecureRandom random);

In case the client does not explicitly initialize the KeyGenerator (via a call to an init method), each provider must supply (and document) a default initialization.

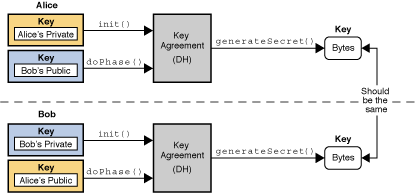
### Creating a Key

The following method generates a secret key:

public SecretKey generateKey();

## The KeyAgreement Class

Key agreement is a protocol by which 2 or more parties can establish the same cryptographic keys, without having to exchange any secret information.



[Description of Figure 14: The KeyAgreement Class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "keyagreement)

Each party initializes their key agreement object with their private key, and then enters the public keys for each party that will participate in the communication. In most cases, there are just two parties, but algorithms such as Diffie-Hellman allow for multiple parties (3 or more) to participate. When all the public keys have been entered, each KeyAgreement object will generate (agree upon) the same key.

The KeyAgreement class provides the functionality of a key agreement protocol. The keys involved in establishing a shared secret are created by one of the key generators (KeyPairGenerator or KeyGenerator), a KeyFactory, or as a result from an intermediate phase of the key agreement protocol.

### Creating a KeyAgreement Object

Each party involved in the key agreement has to create a KeyAgreement object. KeyAgreement objects are obtained by using one of the KeyAgreement [getInstance() static factory methods](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderImplReq).

### Initializing a KeyAgreement Object

You initialize a KeyAgreement object with your private information. In the case of Diffie-Hellman, you initialize it with your Diffie-Hellman private key. Additional initialization information may contain a source of randomness and/or a set of algorithm parameters. Note that if the requested key agreement algorithm requires the specification of algorithm parameters, and only a key, but no parameters are provided to initialize the KeyAgreement object, the key must contain the required algorithm parameters. (For example, the Diffie-Hellman algorithm uses a prime modulus p and a base generator g as its parameters.)

To initialize a KeyAgreement object, call one of its init methods:

public void init(Key key);

public void init(Key key, SecureRandom random);

public void init(Key key, AlgorithmParameterSpec params);

public void init(Key key, AlgorithmParameterSpec params,

SecureRandom random);

### Executing a KeyAgreement Phase

Every key agreement protocol consists of a number of phases that need to be executed by each party involved in the key agreement.

To execute the next phase in the key agreement, call the doPhase method:

public Key doPhase(Key key, boolean lastPhase);

The key parameter contains the key to be processed by that phase. In most cases, this is the public key of one of the other parties involved in the key agreement, or an intermediate key that was generated by a previous phase. doPhase may return an intermediate key that you may have to send to the other parties of this key agreement, so they can process it in a subsequent phase.

The lastPhase parameter specifies whether or not the phase to be executed is the last one in the key agreement: A value of FALSE indicates that this is not the last phase of the key agreement (there are more phases to follow), and a value of TRUE indicates that this is the last phase of the key agreement and the key agreement is completed, i.e., generateSecret can be called next.

In the example of [Diffie-Hellman between two parties](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#AppD) , you call doPhase once, with lastPhase set to TRUE. In the example of Diffie-Hellman between three parties, you call doPhase twice: the first time with lastPhase set to FALSE, the 2nd time with lastPhase set to TRUE.

### Generating the Shared Secret

After each party has executed all the required key agreement phases, it can compute the shared secret by calling one of the generateSecret methods:

public byte[] generateSecret();

public int generateSecret(byte[] sharedSecret, int offset);

public SecretKey generateSecret(String algorithm);