# **Java KeyPairGenerator**

The *Java KeyPairGenerator* class (java.security.KeyPairGenerator) is used to generate asymmetric encryption / decryption key pairs. An asymmetric key pair consists of two keys. The first key is typically used to encrypt data. The second key which is used to decrypt data encrypted with the first key.

**Public Key, Private Key Type Key Pairs**

The most commonly known type of asymmetric key pair is the *public key, private key* type of key pair. The private key is used to encrypt data, and the public key can be used to decrypt the data again. Actually, you could also encrypt data using the public key and decrypt it using the private key.

The private key is normally kept secret, and the public key can be made publicly available. Thus, if Jack encrypts some data with his private key, everyone in possession of Jack's public key can decrypt it.

**Creating a KeyPairGenerator Instance**

To use the Java KeyPairGenerator you must first create a KeyPairGenerator instance. Creating a KeyPairGenerator instance is done by calling the method getInstance() method. Here is an example of creating a Java KeyPairGenerator instance:

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");

The getInstance() method takes the name of the encryption algorithm to generate the key pair for. In this example we use the name RSA.

**Initializing the KeyPairGenerator**

Depending on the algorithm the key pair is generated for, you may have to initialize the KeyPairGeneratorinstance. Initializing the KeyPairGenerator is done by calling its initialize() method. Here is an example of initializing a Java KeyPairGenerator instance:

keyPairGenerator.initialize(2048);

This example initializes the KeyPairGenerator to generate keys of 2048 bits in size.

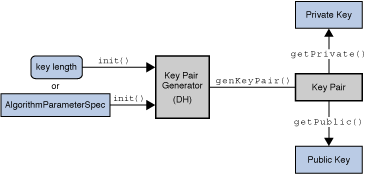
**Generating a Key Pair**

To generate a KeyPair with a KeyPairGenerator you call the generateKeyPair() method. Here is an example of generating a KeyPair with the KeyPairGenerator:

KeyPair keyPair = keyPairGenerator.generateKeyPair();

## The KeyPairGenerator Class

The KeyPairGenerator class is an [engine class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Engine) used to generate pairs of public and private keys.



[Description of Figure 12: The KeyPairGenerator Class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "keypairgenerator)

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

Please see the [Examples](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KPGEx) section for examples of calls to the methods documented below.

### Creating a KeyPairGenerator

All key pair generation starts with a KeyPairGenerator. KeyPairGenerator objects are obtained by using one of the KeyPairGenerator [getInstance() static factory methods](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderImplReq).

### Initializing a KeyPairGenerator

A key pair generator for a particular algorithm creates a public/private key pair that can be used with this algorithm. It also associates algorithm-specific parameters with each of the generated keys.

A key pair generator needs to be initialized before it can generate keys. In most cases, algorithm-independent initialization is sufficient. But in other cases, algorithm-specific initialization can be used.

#### Algorithm-Independent Initialization

All key pair generators share the concepts of a keysize and a source of randomness. The keysize is interpreted differently for different algorithms. For example, in the case of the DSA algorithm, the keysize corresponds to the length of the modulus. (See the [Standard Names](https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html) document for information about the keysizes for specific algorithms.)

An initialize method takes two universally shared types of arguments:

void initialize(int keysize, SecureRandom random)

Another initialize method takes only a keysize argument; it uses a system-provided source of randomness:

void initialize(int keysize)

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

If the algorithm is a "DSA" algorithm, and the modulus size (keysize) is 512, 768, or 1024, then the SUN provider uses a set of precomputed values for the p, q, and g parameters. If the modulus size is not one of the above values, the SUN provider creates a new set of parameters. Other providers might have precomputed parameter sets for more than just the three modulus sizes mentioned above. Still others might not have a list of precomputed parameters at all and instead always create new parameter sets.

#### Algorithm-Specific Initialization

For situations where a set of algorithm-specific parameters already exists (such as "community parameters" in DSA), there are two initialize methods that have an [AlgorithmParameterSpec](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#AlgorithmParameterSpec) argument. One also has a SecureRandom argument, while the source of randomness is system-provided for the other:

void initialize(AlgorithmParameterSpec params,

SecureRandom random)

void initialize(AlgorithmParameterSpec params)

See the [Examples](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KPGEx) section for more details.