# **Java KeyPair**

The *Java KeyPair* class (java.security.KeyPair represents an asymmetric key pair. In other words, a public key + private key pair. A KeyPair instance is typically used when performing asymmetric cryptography, like encrypting or signing data.

**Obtaining a KeyPair Instance**

You will normally obtain a KeyPair instance from a Java **[keystore](http://tutorials.jenkov.com/java-cryptography/keystore.html)** or a Java **[KeyPairGenerator](http://tutorials.jenkov.com/java-cryptography/keypairgenerator.html)**.

**Accessing KeyPair Public Key**

You can access the PublicKey of a KeyPair by calling its getPublic() method. Her is an example of obtaining the PublicKey from a KeyPair instance:

PublicKey publicKey = keyPair.getPublic();

**Accessing the KeyPair Private Key**

You can access the PrivateKey of a KeyPair by calling the getPrivate() method. Here is an example of obtaining the PrivateKey from a KeyPair instance:

PrivateKey privateKey = keyPair.getPrivate();

## Key Interfaces

To this point, we have focused the high-level uses of the JCA without getting lost in the details of what keys are and how they are generated/represented. It is now time to turn our attention to keys.

The java.security.Key interface is the top-level interface for all opaque keys. It defines the functionality shared by all opaque key objects.

An *opaque* key representation is one in which you have no direct access to the key material that constitutes a key. In other words: "opaque" gives you limited access to the key--just the three methods defined by the Key interface (see below): getAlgorithm, getFormat, and getEncoded.

This is in contrast to a *transparent* representation, in which you can access each key material value individually, through one of the get methods defined in the corresponding [specification class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeySpec).

All opaque keys have three characteristics:

**An Algorithm**

The key algorithm for that key. The key algorithm is usually an encryption or asymmetric operation algorithm (such as AES, DSA or RSA), which will work with those algorithms and with related algorithms (such as SHA256WithRSA) The name of the algorithm of a key is obtained using this method:

String getAlgorithm()

**An Encoded Form**

The external encoded form for the key used when a standard representation of the key is needed outside the Java Virtual Machine, as when transmitting the key to some other party. The key is encoded according to a standard format (such as X.509 or PKCS8), and is returned using the method:

byte[] getEncoded()

**A Format**

The name of the format of the encoded key. It is returned by the method:

String getFormat()

Keys are generally obtained through key generators such as [KeyGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeyGenerator) and [KeyPairGenerator](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeyPairGenerator), certificates, [key specifications](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeySpec) (using a [KeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeyFactory)), or a [KeyStore](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyStore) implementation accessing a keystore database used to manage keys. It is possible to parse encoded keys, in an algorithm-dependent manner, using a [KeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeyFactory).

It is also possible to parse certificates, using a [CertificateFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "CertificateFactory).

Here is a list of interfaces which extend the Key interface in the java.security.interfaces and javax.crypto.interfaces packages:

|  |  |  |
| --- | --- | --- |
| * [SecretKey](https://docs.oracle.com/javase/7/docs/api/javax/crypto/SecretKey.html) * [PBEKey](https://docs.oracle.com/javase/7/docs/api/javax/crypto/interfaces/PBEKey.html) | * [PrivateKey](https://docs.oracle.com/javase/7/docs/api/java/security/PrivateKey.html) * [DHPrivateKey](https://docs.oracle.com/javase/7/docs/api/javax/crypto/interfaces/DHPrivateKey.html) * [DSAPrivateKey](https://docs.oracle.com/javase/7/docs/api/java/security/interfaces/DSAPrivateKey.html) * [ECPrivateKey](https://docs.oracle.com/javase/7/docs/api/java/security/interfaces/ECPrivateKey.html) * [RSAMultiPrimePrivateCrtKey](https://docs.oracle.com/javase/7/docs/api/java/security/interfaces/RSAMultiPrimePrivateCrtKey.html) * [RSAPrivateCrtKey](https://docs.oracle.com/javase/7/docs/api/java/security/interfaces/RSAPrivateCrtKey.html) * [RSAPrivateKey](https://docs.oracle.com/javase/7/docs/api/java/security/interfaces/RSAPrivateKey.html) | * [PublicKey](https://docs.oracle.com/javase/7/docs/api/java/security/PublicKey.html) * [DHPublicKey](https://docs.oracle.com/javase/7/docs/api/javax/crypto/interfaces/DHPublicKey.html) * [DSAPublicKey](https://docs.oracle.com/javase/7/docs/api/java/security/interfaces/DSAPublicKey.html) * [ECPublicKey](https://docs.oracle.com/javase/7/docs/api/java/security/interfaces/ECPublicKey.html) * [RSAPublicKey](https://docs.oracle.com/javase/7/docs/api/java/security/interfaces/RSAPublicKey.html) |

### The PublicKey and PrivateKey Interfaces

The PublicKey and PrivateKey interfaces (which both extend the Key interface) are methodless interfaces, used for type-safety and type-identification.

## The KeyPair Class

The KeyPair class is a simple holder for a key pair (a public key and a private key). It has two public methods, one for returning the private key, and the other for returning the public key:

PrivateKey getPrivate()

PublicKey getPublic()

## Key Specification Interfaces and Classes

Key objects and key specifications (KeySpecs) are two different representations of key data. Ciphers use Key objects to initialize their encryption algorithms, but keys may need to be converted into a more portable format for transmission or storage.

A *transparent* representation of keys means that you can access each key material value individually, through one of the get methods defined in the corresponding specification class. For example, DSAPrivateKeySpec defines getX, getP, getQ, and getG methods, to access the private key x, and the DSA algorithm parameters used to calculate the key: the prime p, the sub-prime q, and the base g. If the key is stored on a hardware device, its specification may contain information that helps identify the key on the device.

This representation is contrasted with an *opaque* representation, as defined by the [Key](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Key) interface, in which you have no direct access to the key material fields. In other words, an "opaque" representation gives you limited access to the key--just the three methods defined by the Key interface: getAlgorithm, getFormat, andgetEncoded.

A key may be specified in an algorithm-specific way, or in an algorithm-independent encoding format (such as ASN.1). For example, a DSA private key may be specified by its components x, p, q, and g (see [DSAPrivateKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/DSAPrivateKeySpec.html)), or it may be specified using its DER encoding (see [PKCS8EncodedKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/PKCS8EncodedKeySpec.html)).

The [KeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeyFactory) and [SecretKeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "SecretKeyFactory) classes can be used to convert between opaque and transparent key representations (that is, between Keys and KeySpecs, assuming that the operation is possible. (For example, private keys on smart cards might not be able leave the card. Such Keys are not convertible.)

In the following sections, we discuss the key specification interfaces and classes in the java.security.spec package.

### The KeySpec Interface

This interface contains no methods or constants. Its only purpose is to group and provide type safety for all key specifications. All key specifications must implement this interface.

### The KeySpec Subinterfaces

Like the Key interface, there are a similar set of KeySpec interfaces.

|  |  |  |
| --- | --- | --- |
| * [SecretKeySpec](https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/SecretKeySpec.html) * [EncodedKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/EncodedKeySpec.html) * [PKCS8EncodedKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/PKCS8EncodedKeySpec.html) * [X509EncodedKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/X509EncodedKeySpec.html) * [DESKeySpec](https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/DESKeySpec.html) * [DESedeKeySpec](https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/DESedeKeySpec.html) * [PBEKeySpec](https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/PBEKeySpec.html) | * [DHPrivateKeySpec](https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/DHPrivateKeySpec.html) * [DSAPrivateKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/DSAPrivateKeySpec.html) * [ECPrivateKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/ECPrivateKeySpec.html) * [RSAMultiPrimePrivateCrtKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/RSAMultiPrimePrivateCrtKeySpec.html) * [RSAPrivateCrtKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/RSAPrivateCrtKeySpec.html) * [RSAPrivateKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/RSAPrivateKeySpec.html) | * [DHPublicKeySpec](https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/DHPublicKeySpec.html) * [DSAPublicKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/DSAPublicKeySpec.html) * [ECPublicKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/ECPublicKeySpec.html) * [RSAPublicKeySpec](https://docs.oracle.com/javase/7/docs/api/java/security/spec/RSAPublicKeySpec.html) |

### The EncodedKeySpec Class

This abstract class (which implements the [KeySpec](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeySpec) interface) represents a public or private key in encoded format. Its getEncoded method returns the encoded key:

abstract byte[] getEncoded();

and its getFormat method returns the name of the encoding format:

abstract String getFormat();

See the next sections for the concrete implementations PKCS8EncodedKeySpec and X509EncodedKeySpec.

#### The PKCS8EncodedKeySpec Class

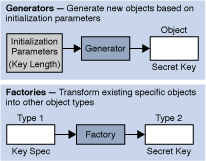
This class, which is a subclass of EncodedKeySpec, represents the DER encoding of a private key, according to the format specified in the PKCS8 standard. Its getEncoded method returns the key bytes, encoded according to the PKCS8 standard. Its getFormat method returns the string "PKCS#8".

#### The X509EncodedKeySpec Class

This class, which is a subclass of EncodedKeySpec, represents the DER encoding of a public key, according to the format specified in the X.509 standard. Its getEncoded method returns the key bytes, encoded according to the X.509 standard. Its getFormat method returns the string "X.509".

## Of Generators and Factories

Newcomers to Java and the JCA APIs in particular sometimes do not grasp the distinction between generators and factories.



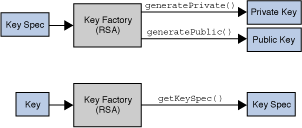
[Description of Figure 9: Differences Between Generators and Factories](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "differences-between-generators-and-factories)

Generators are used to **generate brand new objects**. Generators can be initialized in either an algorithm-dependent or algorithm-independent way. For example, to create a Diffie-Hellman (DH) keypair, an application could specify the necessary P and G values, or the generator could simply be initialized with the appropriate key length, and the generator will select appropriate P and G values. In both cases, the generator will produce brand new keys based on the parameters.

On the other hand, factories are used to **convert data from one existing object type to another**. For example, an application might have available the components of a DH private key and can package them as a [KeySpec](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeySpec), but needs to convert them into a [PrivateKey](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "Key) object that can be used by a KeyAgreement object, or vice-versa. Or they might have the byte array of a certificate, but need to use a CertificateFactory to convert it into a X509Certificate object. Applications use factory objects to do the conversion.

## The KeyFactory Class

The KeyFactory class is an [engine class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Engine) designed to perform conversions between opaque cryptographic [Keys](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Key) and [key specifications](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeySpecs) (transparent representations of the underlying key material).



[Description of Figure 10: The KeyFactory Class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "keyfactory)

Key factories are bidirectional. They allow you to build an opaque key object from a given key specification (key material), or to retrieve the underlying key material of a key object in a suitable format.

Multiple compatible key specifications can exist for the same key. For example, a DSA public key may be specified by its components y, p, q, and g (see java.security.spec.DSAPublicKeySpec), or it may be specified using its DER encoding according to the X.509 standard (see [X509EncodedKeySpec](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#X509EncodedKeySpec)).

A key factory can be used to translate between compatible key specifications. Key parsing can be achieved through translation between compatible key specifications, e.g., when you translate from X509EncodedKeySpec to DSAPublicKeySpec, you basically parse the encoded key into its components. For an example, see the end of the [Generating/Verifying Signatures Using Key Specifications and KeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyFactoryEx) section.

### Creating a KeyFactory Object

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

### Converting Between a Key Specification and a Key Object

If you have a key specification for a public key, you can obtain an opaque PublicKey object from the specification by using the generatePublic method:

PublicKey generatePublic(KeySpec keySpec)

Similarly, if you have a key specification for a private key, you can obtain an opaque PrivateKey object from the specification by using the generatePrivate method:

PrivateKey generatePrivate(KeySpec keySpec)

### Converting Between a Key Object and a Key Specification

If you have a Key object, you can get a corresponding key specification object by calling the getKeySpec method:

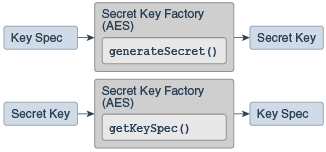
KeySpec getKeySpec(Key key, Class keySpec)

keySpec identifies the specification class in which the key material should be returned. It could, for example, be DSAPublicKeySpec.class, to indicate that the key material should be returned in an instance of the DSAPublicKeySpec class.

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

## The SecretKeyFactory Class

This class represents a factory for secret keys. Unlike [KeyFactory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html" \l "KeyFactory), a javax.crypto.SecretKeyFactory object operates only on secret (symmetric) keys, whereas a java.security.KeyFactory object processes the public and private key components of a key pair.



[Description of Figure 11: The SecretKeyFactory Class](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "secretkeyfactory)

Key factories are used to convert [Keys](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Key) (opaque cryptographic keys of type java.security.Key) into [key specifications](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeySpecs) (transparent representations of the underlying key material in a suitable format), and vice versa.

Objects of type java.security.Key, of which java.security.PublicKey, java.security.PrivateKey, and javax.crypto.SecretKey are subclasses, are opaque key objects, because you cannot tell how they are implemented. The underlying implementation is provider-dependent, and may be software or hardware based. Key factories allow providers to supply their own implementations of cryptographic keys.

For example, if you have a key specification for a Diffie Hellman public key, consisting of the public value y, the prime modulus p, and the base g, and you feed the same specification to Diffie-Hellman key factories from different providers, the resulting PublicKey objects will most likely have different underlying implementations.

A provider should document the key specifications supported by its secret key factory. For example, the SecretKeyFactory for DES keys supplied by the SunJCE provider supports DESKeySpec as a transparent representation of DES keys, the SecretKeyFactory for DES-EDE keys supports DESedeKeySpec as a transparent representation of DES-EDE keys, and the SecretKeyFactory for PBE supports PBEKeySpec as a transparent representation of the underlying password.

The following is an example of how to use a SecretKeyFactory to convert secret key data into a SecretKey object, which can be used for a subsequent Cipher operation:

// Note the following bytes are not realistic secret key data

// bytes but are simply supplied as an illustration of using data

// bytes (key material) you already have to build a DESedeKeySpec.

byte[] desEdeKeyData = getKeyData();

DESedeKeySpec desEdeKeySpec = new DESedeKeySpec(desEdeKeyData);

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DESede");

SecretKey secretKey = keyFactory.generateSecret(desEdeKeySpec);

In this case, the underlying implementation of SecretKey is based on the provider of KeyFactory.

An alternative, provider-independent way of creating a functionally equivalent SecretKey object from the same key material is to use the javax.crypto.spec.SecretKeySpec class, which implements the javax.crypto.SecretKey interface:

byte[] aesKeyData = getKeyData();

SecretKeySpec secretKey = new SecretKeySpec(aesKeyData, "AES");

### Creating a SecretKeyFactory Object

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

### Converting Between a Key Specification and a Secret Key Object

If you have a key specification for a secret key, you can obtain an opaque SecretKey object from the specification by using the generateSecret method:

SecretKey generateSecret(KeySpec keySpec)

### Converting Between a Secret Key Object and a Key Specification

If you have a Secret Key object, you can get a corresponding key specification object by calling the getKeySpec method:

KeySpec getKeySpec(Key key, Class keySpec)

keySpec identifies the specification class in which the key material should be returned. It could, for example, be DESKeySpec.class, to indicate that the key material should be returned in an instance of the DESKeySpec class.

### Generating a Key Pair

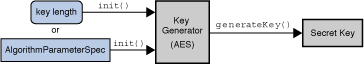
The procedure for generating a key pair is always the same, regardless of initialization (and of the algorithm). You always call the following method from KeyPairGenerator:

KeyPair generateKeyPair()

Multiple calls to generateKeyPair will yield different key pairs.

## 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" \l "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.