



CHAPTER 18

javax.crypto and Subpackages

This chapter documents the cryptographic features (including encryption and decryption) of the `javax.crypto` package and its subpackages. These packages were originally part of the Java Cryptography Extension (JCE) before being integrated into Java 1.4, which is why they have the “javax” extension prefix. All of the commonly used cryptography classes are in the `javax.crypto` package itself. The `javax.crypto.interfaces` subpackage defines algorithm-specific interfaces for certain type of cryptographic keys. The `javax.crypto.spec` subpackage defines classes that provide a transparent, portable, and provider-independent representation of cryptographic keys and related objects.

Package `javax.crypto`

Java 1.4

The `javax.crypto` package defines classes and interfaces for various cryptographic operations. The central class is `Cipher`, which is used to encrypt and decrypt data. `CipherInputStream` and `CipherOutputStream` are utility classes that use a `Cipher` object to encrypt or decrypt streaming data. `SealedObject` is another important utility class that uses a `Cipher` object to encrypt an arbitrary serializable Java object.

The `KeyGenerator` class creates the `SecretKey` objects used by `Cipher` for encryption and decryption. `SecretKeyFactory` encodes and decodes `SecretKey` objects. The `KeyAgreement` class enables two or more parties to agree on a `SecretKey` in such a way that an eavesdropper cannot determine the key. The `Mac` class computes a message authentication code (MAC) that can ensure the integrity of a transmission between two parties who share a `SecretKey`. A MAC is akin to a digital signature, except that it is based on a secret key instead of a public/private key pair.

Like the `java.security` package, the `javax.crypto` package is provider-based, so that arbitrary cryptographic implementations may be plugged into any Java installation. Various classes in this package have names that end in `Spi`. These classes define a service-provider interface and must be implemented by each cryptographic provider that wishes to provide an implementation of a particular cryptographic service or algorithm.

This package was originally shipped as part of the Java Cryptography Extension (JCE), but it has been added to the core platform in Java 1.4. A version of the JCE is still available (see <http://java.sun.com/security/>) as a standard extension for Java 1.2 and Java 1.3. This package is distributed with a cryptographic provider named “SunJCE” that

includes a robust set of implementations for Cipher, KeyAgreement, Mac, and other classes. This provider is installed by the default java.security properties in Java 1.4 distributions.

A full tutorial on cryptography is beyond the scope of this chapter and of this book. In order to use this package, you need to have a basic understanding of cryptographic algorithms such as DES. In order to take full advantage of this package, you also need to have a detailed understanding of things like feedback modes, padding schemes, the Diffie-Hellman key-agreement protocol, and so on. For a good introduction to modern cryptography in Java, see *Java Cryptography* (O'Reilly). For more in-depth coverage, not specific to Java, see *Applied Cryptography* (Wiley).

Interface:

public interface **SecretKey** extends java.security.Key;

Classes:

```
public class Cipher;
    public class NullCipher extends Cipher;
public class CipherInputStream extends java.io.FilterInputStream;
public class CipherOutputStream extends java.io.FilterOutputStream;
public abstract class CipherSpi;
public class EncryptedPrivateKeyInfo;
public class ExemptionMechanism;
public abstract class ExemptionMechanismSpi;
public class KeyAgreement;
public abstract class KeyAgreementSpi;
public class KeyGenerator;
public abstract class KeyGeneratorSpi;
public class Mac implements Cloneable;
public abstract class MacSpi;
public class SealedObject implements Serializable;
public class SecretKeyFactory;
public abstract class SecretKeyFactorySpi;
```

Exceptions:

```
public class BadPaddingException extends java.security.GeneralSecurityException;
public class ExemptionMechanismException extends java.security.GeneralSecurityException;
public class IllegalBlockSizeException extends java.security.GeneralSecurityException;
public class NoSuchPaddingException extends java.security.GeneralSecurityException;
public class ShortBufferException extends java.security.GeneralSecurityException;
```

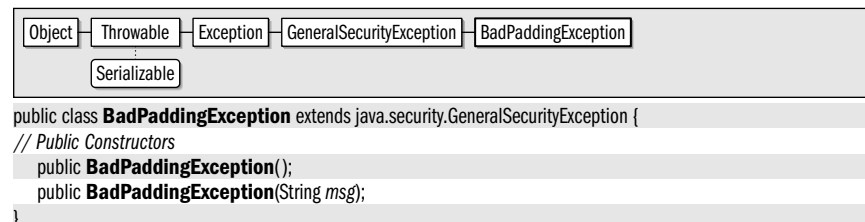
BadPaddingException

Java 1.4

javax.crypto

serializable checked

This exception signals that input data to a Cipher is not padded correctly.



BadPaddingException

Thrown By: Cipher.doFinal(), CipherSpi.engineDoFinal(), SealedObject.getObject()

Cipher

Java 1.4

javax.crypto

This class performs encryption and decryption of byte arrays. **Cipher** is provider-based, so to obtain a **Cipher** object, you must call the static `getInstance()` factory method. The arguments to this method are a string that describes the type of encryption desired and, optionally, the name of the provider whose implementation should be used. To specify the desired type of encryption, you can simply specify the name of an encryption algorithm, such as "DES". Or you can specify a three-part name that includes the encryption algorithm, the algorithm operating mode, and the padding scheme. These three parts are separated by slash characters, as in "DES/CBC/PKCS5Padding". Finally, if you are requesting a block cipher algorithm in a stream mode, you can specify the number of bits to be processed at a time by following the name of the feedback mode with a number of bits. For example: "DES/CFB8/NoPadding".

The "SunJCE" provider supports the following cryptographic algorithms:

"DES"

The Digital Encryption Standard.

"DESede"

Triple DES encryption, also known as "TripleDES".

"Blowfish"

The Blowfish block cipher designed by Bruce Schneier.

"PBEWithMD5AndDES"

A password-based encryption scheme specified in PKCS#5. This algorithm implicitly uses "CBC" mode and the "PKCS5Padding" padding; it cannot be used with other modes or padding schemes.

"PBEWithMD5AndTripleDES"

A password-based encryption similar to "PBEWithMD5AndDES", but uses DESede instead of DES.

SunJCE supports the following operating modes:

"ECB"

Electronic Codebook mode

"CBC"

Cipher Block Chaining mode

"CFB"

Cipher Feedback mode

"OFB"

Output Feedback mode

"PCBC"

Plaintext Cipher Block Chaining mode

Finally, the "SunJCE" provider also supports two padding schemes: "NoPadding" and "PKCS5Padding". The name "SSL3Padding" is reserved, but this padding scheme is not implemented in the current release of "SunJCE".

Once you have obtained a **Cipher** object for the desired cryptographic algorithm, mode, and padding scheme, you must initialize it by calling one of the `init()` methods. The first

argument to `init()` is one of the constants `ENCRYPT_MODE` or `DECRYPT_MODE`. The second argument is a `java.security.Key` object that performs the encryption or decryption. If you use one of the symmetric (i.e., non-public key) encryption algorithms supported by the “SunJCE” provider, this `Key` object is a `SecretKey` implementation. You can optionally pass a `java.security.SecureRandom` object to `init()` to provide a source of randomness. If you do not, the `Cipher` implementation provides its own pseudo-random number generator.

Some cryptographic algorithms require additional initialization parameters; these can be passed to `init()` as a `java.security.AlgorithmParameters` object or as a `java.security.spec.AlgorithmParameterSpec` object. When encrypting, you can omit these parameters, and the `Cipher` implementation uses default values or generates appropriate random parameters for you. In this case, you should call `getParameters()` after performing encryption to obtain the `AlgorithmParameters` used to encrypt. These parameters are required in order to decrypt, and must therefore be saved or transferred along with the encrypted data. Of the algorithms supported by the “SunJCE” provider, the block ciphers “DES”, “DESEde”, and “Blowfish” all require an initialization vector when they are used in “CBC”, “CFB”, “OFB”, or “PCBC” mode. You can represent an initialization vector with a `javax.crypto.spec.IvParameterSpec` object and obtain the raw bytes of the initialization vector used by a `Cipher` with the `getIV()` method. The “PBESWithMD5AndDES” algorithm requires a salt and iteration count as parameters. These can be specified with a `javax.crypto.spec.PBEParameterSpec` object.

Once you have obtained and initialized a `Cipher` object, you are ready to use it for encryption or decryption. If you have only a single array of bytes to encrypt or decrypt, pass that input array to one of the `doFinal()` methods. Some versions of this method return the encrypted or decrypted bytes as the return value of the function. Other versions store the encrypted or decrypted bytes to another byte array you specify. If you choose to use one of these latter methods, you should first call `getOutputSize()` to determine the required size of the output array. If you want to encrypt or decrypt data from a streaming source or have more than one array of data, pass the data to one of the `update()` methods, calling it as many times as necessary. Then pass the last array of data to one of the `doFinal()` methods. If you are working with streaming data, consider using the `CipherInputStream` and `CipherOutputStream` classes instead.

```
public class Cipher {
    // Protected Constructors
    protected Cipher(CipherSpi cipherSpi, java.security.Provider provider, String transformation);

    // Public Constants
    public static final int DECRYPT_MODE;           =2
    public static final int ENCRYPT_MODE;          =1
    public static final int PRIVATE_KEY;          =2
    public static final int PUBLIC_KEY;           =1
    public static final int SECRET_KEY;           =3
    public static final int UNWRAP_MODE;          =4
    public static final int WRAP_MODE;            =3

    // Public Class Methods
    public static final Cipher getInstance(String transformation) throws java.security.NoSuchAlgorithmException,
        NoSuchPaddingException;
    public static final Cipher getInstance(String transformation, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException,
        NoSuchPaddingException;
    public static final Cipher getInstance(String transformation, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException, NoSuchPaddingException;

    // Property Accessor Methods (by property name)
    public final String getAlgorithm();
    public final int getBlockSize();
}
```

Cipher

```
public final ExemptionMechanism getExemptionMechanism();
public final byte[] getIV();
public final java.security.AlgorithmParameters getParameters();
public final java.security.Provider getProvider();
// Public Instance Methods
public final byte[] doFinal() throws IllegalStateException, IllegalBlockSizeException, BadPaddingException;
public final byte[] doFinal(byte[] input) throws IllegalStateException, IllegalBlockSizeException,
    BadPaddingException;
public final int doFinal(byte[] output, int outputOffset) throws IllegalStateException, IllegalBlockSizeException,
    ShortBufferException, BadPaddingException;
public final byte[] doFinal(byte[] input, int inputOffset, int inputLen) throws IllegalStateException,
    IllegalBlockSizeException, BadPaddingException;
public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output) throws IllegalStateException,
    ShortBufferException, IllegalBlockSizeException, BadPaddingException;
public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)
    throws IllegalStateException, ShortBufferException, IllegalBlockSizeException, BadPaddingException;
public final int getOutputSize(int inputLen) throws IllegalStateException;
public final void init(int opmode, java.security.cert.Certificate certificate) throws java.security.InvalidKeyException;
public final void init(int opmode, java.security.Key key) throws java.security.InvalidKeyException;
public final void init(int opmode, java.security.cert.Certificate certificate, java.security.SecureRandom random)
    throws java.security.InvalidKeyException;
public final void init(int opmode, java.security.Key key, java.security.AlgorithmParameters params)
    throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;
public final void init(int opmode, java.security.Key key, java.security.SecureRandom random)
    throws java.security.InvalidKeyException;
public final void init(int opmode, java.security.Key key, java.security.spec.AlgorithmParameterSpec params)
    throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;
public final void init(int opmode, java.security.Key key, java.security.spec.AlgorithmParameterSpec params,
    java.security.SecureRandom random) throws java.security.InvalidKeyException,
    java.security.InvalidAlgorithmParameterException;
public final void init(int opmode, java.security.Key key, java.security.AlgorithmParameters params,
    java.security.SecureRandom random) throws java.security.InvalidKeyException,
    java.security.InvalidAlgorithmParameterException;
public final java.security.Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType)
    throws IllegalStateException, java.security.InvalidKeyException, java.security.NoSuchAlgorithmException;
public final byte[] update(byte[] input) throws IllegalStateException;
public final byte[] update(byte[] input, int inputOffset, int inputLen) throws IllegalStateException;
public final int update(byte[] input, int inputOffset, int inputLen, byte[] output) throws IllegalStateException,
    ShortBufferException;
public final int update(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)
    throws IllegalStateException, ShortBufferException;
public final byte[] wrap(java.security.Key key) throws IllegalStateException, IllegalBlockSizeException,
    java.security.InvalidKeyException;
}
```

Subclasses: NullCipher

Passed To: CipherInputStream.CipherInputStream(), CipherOutputStream.CipherOutputStream(),
EncryptedPrivateKeyInfo.getKeySpec(), SealedObject.{getObject(), SealedObject()}

Returned By: Cipher.getInstance()

CipherInputStream**Java 1.4****javax.crypto**

This class is an input stream that uses a **Cipher** object to encrypt or decrypt the bytes it reads from another stream. You must initialize the **Cipher** object before passing it to the **CipherInputStream()** constructor.

Object	InputStream	FilterInputStream	CipherInputStream
--------	-------------	-------------------	-------------------

```

public class CipherInputStream extends java.io.FilterInputStream {
// Public Constructors
    public CipherInputStream(java.io.InputStream is, Cipher c);
// Protected Constructors
    protected CipherInputStream(java.io.InputStream is);
// Public Methods Overriding FilterInputStream
    public int available() throws java.io.IOException;
    public void close() throws java.io.IOException;
    public boolean markSupported(); constant
    public int read() throws java.io.IOException;
    public int read(byte[] b) throws java.io.IOException;
    public int read(byte[] b, int off, int len) throws java.io.IOException;
    public long skip(long n) throws java.io.IOException;
}

```

CipherOutputStream**Java 1.4****javax.crypto**

This class is an output stream that uses a **Cipher** object to encrypt or decrypt bytes before passing them to another output stream. You must initialize the **Cipher** object before passing it to the **CipherOutputStream()** constructor. If you are using a **Cipher** with any kind of padding, you must not call **flush()** until you are done writing all data to the stream; otherwise decryption fails.

Object	OutputStream	FilterOutputStream	CipherOutputStream
--------	--------------	--------------------	--------------------

```

public class CipherOutputStream extends java.io.FilterOutputStream {
// Public Constructors
    public CipherOutputStream(java.io.OutputStream os, Cipher c);
// Protected Constructors
    protected CipherOutputStream(java.io.OutputStream os);
// Public Methods Overriding FilterOutputStream
    public void close() throws java.io.IOException;
    public void flush() throws java.io.IOException;
    public void write(int b) throws java.io.IOException;
    public void write(byte[] b) throws java.io.IOException;
    public void write(byte[] b, int off, int len) throws java.io.IOException;
}

```

CipherSpi**Java 1.4****javax.crypto**

This abstract class defines the service-provider interface for **Cipher**. A cryptographic provider must implement a concrete subclass of this class for each encryption algorithm it supports. A provider can implement a separate class for each combination of algorithm, mode, and padding scheme it supports or implement more general classes and

CipherSpi

leave the mode and/or padding scheme to be specified in calls to `engineSetMode()` and `engineSetPadding()`. Applications never need to use or subclass this class.

```
public abstract class CipherSpi {  
    // Public Constructors  
    public CipherSpi();  
    // Protected Instance Methods  
    protected abstract byte[] engineDoFinal(byte[] input, int inputOffset, int inputLen)  
        throws IllegalBlockSizeException, BadPaddingException;  
    protected abstract int engineDoFinal(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)  
        throws ShortBufferException, IllegalBlockSizeException, BadPaddingException;  
    protected abstract int engineGetBlockSize();  
    protected abstract byte[] engineGetIV();  
    protected int engineGetKeySize(java.security.Key key) throws java.security.InvalidKeyException;  
    protected abstract int engineGetOutputSize(int inputLen);  
    protected abstract java.security.AlgorithmParameters engineGetParameters();  
    protected abstract void engineInit(int opmode, java.security.Key key, java.security.SecureRandom random)  
        throws java.security.InvalidKeyException;  
    protected abstract void engineInit(int opmode, java.security.Key key, java.security.AlgorithmParameters params,  
        java.security.SecureRandom random) throws java.security.InvalidKeyException,  
        java.security.InvalidAlgorithmParameterException;  
    protected abstract void engineInit(int opmode, java.security.Key key,  
        java.security.spec.AlgorithmParameterSpec params,  
        java.security.SecureRandom random) throws java.security.InvalidKeyException,  
        java.security.InvalidAlgorithmParameterException;  
    protected abstract void engineSetMode(String mode) throws java.security.NoSuchAlgorithmException;  
    protected abstract void engineSetPadding(String padding) throws NoSuchPaddingException;  
    protected java.security.Key engineUnwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType)  
        throws java.security.InvalidKeyException, java.security.NoSuchAlgorithmException;  
    protected abstract byte[] engineUpdate(byte[] input, int inputOffset, int inputLen);  
    protected abstract int engineUpdate(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)  
        throws ShortBufferException;  
    protected byte[] engineWrap(java.security.Key key) throws IllegalBlockSizeException,  
        java.security.InvalidKeyException;  
}
```

Passed To: `Cipher.Cipher()`

EncryptedPrivateKeyInfo

Java 1.4

`javax.crypto`

This class represents an encrypted private key. `getEncryptedData()` returns the encrypted bytes. `getAlgName()` and `getAlgParameters()` return the algorithm name and parameters used to encrypt it. Pass a `Cipher` object to `getKeySpec()` to decrypt the key.

```
public class EncryptedPrivateKeyInfo {  
    // Public Constructors  
    public EncryptedPrivateKeyInfo(byte[] encoded) throws java.io.IOException;  
    public EncryptedPrivateKeyInfo(java.security.AlgorithmParameters algParams, byte[] encryptedData)  
        throws java.security.NoSuchAlgorithmException;  
    public EncryptedPrivateKeyInfo(String algName, byte[] encryptedData)  
        throws java.security.NoSuchAlgorithmException;  
    // Public Instance Methods  
    public String getAlgName();  
    public java.security.AlgorithmParameters getAlgParameters();  
}
```

```

    public byte[] getEncoded() throws java.io.IOException;
    public byte[] getEncryptedData();
    public java.security.spec.PKCS8EncodedKeySpec getKeySpec(Cipher c)
        throws java.security.spec.InvalidKeySpecException;
}

```

ExemptionMechanism

Java 1.4

javax.crypto

Some countries place legal restrictions on the use of cryptographic algorithms. In some cases, a program may be exempt from these restrictions if it implements an “exemption mechanism” such as key recovery, key escrow, or key weakening. This class defines a general API to such mechanism. This class is rarely used and is not supported in the default implementation provided by Sun. Using this class successfully is quite complex and is beyond the scope of this reference. For details, see the discussion “How to Make Applications ‘Exempt’ from Cryptographic Restrictions” in the *JCE Reference Guide*, which is part of the standard bundle of documentation shipped by Sun with the JDK.

```

public class ExemptionMechanism {
    // Protected Constructors
    protected ExemptionMechanism(ExemptionMechanismSpi exmechSpi, java.security.Provider provider,
        String mechanism);

    // Public Class Methods
    public static final ExemptionMechanism getInstance(String mechanism)
        throws java.security.NoSuchAlgorithmException;
    public static final ExemptionMechanism getInstance(String mechanism, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
    public static final ExemptionMechanism getInstance(String mechanism, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;

    // Public Instance Methods
    public final byte[] genExemptionBlob() throws IllegalStateException, ExemptionMechanismException;
    public final int genExemptionBlob(byte[] output) throws IllegalStateException, ShortBufferException,
        ExemptionMechanismException;
    public final int genExemptionBlob(byte[] output, int outputOffset) throws IllegalStateException,
        ShortBufferException, ExemptionMechanismException;
    public final String getName();
    public final int getOutputSize(int inputLen) throws IllegalStateException;
    public final java.security.Provider getProvider();
    public final void init(java.security.Key key) throws java.security.InvalidKeyException, ExemptionMechanismException;
    public final void init(java.security.Key key, java.security.spec.AlgorithmParameterSpec params)
        throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException,
        ExemptionMechanismException;
    public final void init(java.security.Key key, java.security.AlgorithmParameters params)
        throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException,
        ExemptionMechanismException;
    public final boolean isCryptoAllowed(java.security.Key key) throws ExemptionMechanismException;

    // Protected Methods Overriding Object
    protected void finalize();
}

```

Returned By: Cipher.getExemptionMechanism(), ExemptionMechanism.getInstance()

ExemptionMechanismException

Java 1.4

javax.crypto

serializable checked

This exception signals a problem in one of the `ExemptionMechanism` methods.



```

public class ExemptionMechanismException extends java.security.GeneralSecurityException {
// Public Constructors
    public ExemptionMechanismException();
    public ExemptionMechanismException(String msg);
}
  
```

Thrown By: `ExemptionMechanism`.{`genExemptionBlob()`, `init()`, `isCryptoAllowed()`},
`ExemptionMechanismSpi`.{`engineGenExemptionBlob()`, `engineInit()`}

ExemptionMechanismSpi

Java 1.4

javax.crypto

This abstract class defines the Service Provider Interface for `ExemptionMechanism`. Security providers may implement this interface, but applications never need to use it. Note that the default “SunJCE” provider does not provide an implementation.

```

public abstract class ExemptionMechanismSpi {
// Public Constructors
    public ExemptionMechanismSpi();
// Protected Instance Methods
    protected abstract byte[] engineGenExemptionBlob() throws ExemptionMechanismException;
    protected abstract int engineGenExemptionBlob(byte[] output, int outputOffset) throws ShortBufferException,
        ExemptionMechanismException;
    protected abstract int engineGetOutputSize(int inputLen);
    protected abstract void engineInit(java.security.Key key) throws java.security.InvalidKeyException,
        ExemptionMechanismException;
    protected abstract void engineInit(java.security.Key key, java.security.AlgorithmParameters params)
        throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException,
        ExemptionMechanismException;
    protected abstract void engineInit(java.security.Key key, java.security.spec.AlgorithmParameterSpec params)
        throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException,
        ExemptionMechanismException;
}
  
```

Passed To: `ExemptionMechanism`.`ExemptionMechanism()`

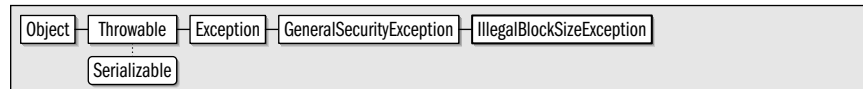
IllegalBlockSizeException

Java 1.4

javax.crypto

serializable checked

This exception signals that the length of data provided to a block cipher (as implemented, for example, by `Cipher` and `SealedObject`) does not match the block size for the cipher.



```

public class IllegalBlockSizeException extends java.security.GeneralSecurityException {
// Public Constructors
  
```

```

public IllegalBlockSizeException();
public IllegalBlockSizeException(String msg);
}

```

Thrown By: Cipher.{doFinal(), wrap()}, CipherSpi.{engineDoFinal(), engineWrap()}, SealedObject.{getObject(), SealedObject()}

KeyAgreement

Java 1.4

javax.crypto

This class provides an API to a key-agreement protocol that allows two or more parties to agree on a secret key without exchanging any secrets and in such a way that an eavesdropper listening in on the communication between those parties cannot determine the secret key. The **KeyAgreement** class is algorithm-independent and provider-based, so you must obtain a **KeyAgreement** object by calling one of the static **getInstance()** factory methods and specifying the name of the desired key agreement algorithm and, optionally, the name of the desired provider of that algorithm. The “SunJCE” provider implements a single key-agreement algorithm named “DiffieHellman”.

To use a **KeyAgreement** object, each party first calls the **init()** method and supplies a **Key** object of its own. Then, each party obtains a **Key** object from one of the other parties to the agreement and calls **doPhase()**. Each party obtains an intermediate **Key** object as the return value of **doPhase()**, and these keys are again exchanged and passed to **doPhase()**. This process typically repeats $n-1$ times, where n is the number of parties, but the actual number of repetitions is algorithm-dependent. When **doPhase()** is called the last time, the second argument must be **true** to indicate that it is the last phase of the agreement. After all calls to **doPhase()** have been made, each party calls **generateSecret()** to obtain an array of bytes or a **SecretKey** object for a named algorithm type. All parties obtain the same bytes or **SecretKey** from this method. The **KeyAgreement** class is not responsible for the transfer of **Key** objects between parties or for mutual authentication among the parties. These tasks must be accomplished through some external mechanism.

The most common type of key agreement is “DiffieHellman” key agreement between two parties. It proceeds as follows. First, both parties obtain a **java.security.KeyPairGenerator** for the “DiffieHellman” algorithm and use it to generate a **java.security.KeyPair** of Diffie-Hellman public and private keys. Each party passes its private key to the **init()** method of its **KeyAgreement** object. (The **init()** method can be passed a **java.security.spec.AlgorithmParameterSpec** object, but the Diffie-Hellman protocol does not require any additional parameters.) Next, the two parties exchange public keys, typically through some kind of networking mechanism (the **KeyAgreement** class is not responsible for the actual exchange of keys). Each party passes the public key of the other party to the **doPhase()** method of its **KeyAgreement** object. There are only two parties to this agreement, so only one phase is required, and the second argument to **doPhase()** is **true**. At this point, both parties call **generateSecret()** to obtain the shared secret key.

A three-party Diffie-Hellman key agreement requires two phases and is slightly more complicated. Let’s call the three parties Alice, Bob, and Carol. Each generates a key pair and uses its private key to initialize its **KeyAgreement** object, as before. Then Alice passes her public key to Bob, Bob passes his to Carol, and Carol passes hers to Alice. Each party passes this public key to **doPhase()**. Since this is not the final **doPhase()**, the second argument is **false**, and **doPhase()** returns an intermediate **Key** object. The three parties exchange these intermediate keys again in the same way: Alice to Bob, Bob to Carol, and Carol to Alice. Now each party passes the intermediate key it has received to **doPhase()** a second time, passing **true** to indicate that this is the final phase. Finally, all three can call **generateSecret()** to obtain a shared key to encrypt future communication.

KeyAgreement

```
public class KeyAgreement {  
    // Protected Constructors  
    protected KeyAgreement(KeyAgreementSpi keyAgreeSpi, java.security.Provider provider, String algorithm);  
    // Public Class Methods  
    public static final KeyAgreement getInstance(String algorithm) throws java.security.NoSuchAlgorithmException;  
    public static final KeyAgreement getInstance(String algorithm, String provider)  
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;  
    public static final KeyAgreement getInstance(String algorithm, java.security.Provider provider)  
        throws java.security.NoSuchAlgorithmException;  
    // Public Instance Methods  
    public final java.security.Key doPhase(java.security.Key key, boolean lastPhase)  
        throws java.security.InvalidKeyException, IllegalStateException;  
    public final byte[] generateSecret() throws IllegalStateException;  
    public final SecretKey generateSecret(String algorithm) throws IllegalStateException,  
        java.security.NoSuchAlgorithmException, java.security.InvalidKeyException;  
    public final int generateSecret(byte[] sharedSecret, int offset) throws IllegalStateException, ShortBufferException;  
    public final String getAlgorithm();  
    public final java.security.Provider getProvider();  
    public final void init(java.security.Key key) throws java.security.InvalidKeyException;  
    public final void init(java.security.Key key, java.security.SecureRandom random)  
        throws java.security.InvalidKeyException;  
    public final void init(java.security.Key key, java.security.spec.AlgorithmParameterSpec params)  
        throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;  
    public final void init(java.security.Key key, java.security.spec.AlgorithmParameterSpec params,  
        java.security.SecureRandom random) throws java.security.InvalidKeyException,  
        java.security.InvalidAlgorithmParameterException;  
}
```

Returned By: KeyAgreement.getInstance()

KeyAgreementSpi

Java 1.4

javax.crypto

This abstract class defines the service-provider interface for **KeyAgreement**. A cryptographic provider must implement a concrete subclass of this class for each encryption algorithm it supports. Applications never need to use or subclass this class.

```
public abstract class KeyAgreementSpi {  
    // Public Constructors  
    public KeyAgreementSpi();  
    // Protected Instance Methods  
    protected abstract java.security.Key engineDoPhase(java.security.Key key, boolean lastPhase)  
        throws java.security.InvalidKeyException, IllegalStateException;  
    protected abstract byte[] engineGenerateSecret() throws IllegalStateException;  
    protected abstract SecretKey engineGenerateSecret(String algorithm) throws IllegalStateException,  
        java.security.NoSuchAlgorithmException, java.security.InvalidKeyException;  
    protected abstract int engineGenerateSecret(byte[] sharedSecret, int offset) throws IllegalStateException,  
        ShortBufferException;  
    protected abstract void engineInit(java.security.Key key, java.security.SecureRandom random)  
        throws java.security.InvalidKeyException;  
    protected abstract void engineInit(java.security.Key key, java.security.spec.AlgorithmParameterSpec params,  
        java.security.SecureRandom random) throws java.security.InvalidKeyException,  
        java.security.InvalidAlgorithmParameterException;  
}
```

Passed To: KeyAgreement.KeyAgreement()

KeyGenerator

Java 1.4

javax.crypto

This class provides an API for generating secret keys for symmetric cryptography. It is similar to `java.security.KeyPairGenerator`, which generates public/private key pairs for asymmetric or public-key cryptography. `KeyGenerator` is algorithm-independent and provider-based, so you must obtain a `KeyGenerator` instance by calling one of the static `getInstance()` factory methods and specifying the name of the cryptographic algorithm for which a key is desired and, optionally, the name of the security provider whose key-generation implementation is to be used. The “SunJCE” provider includes `KeyGenerator` implementations for the “DES”, “DESEde”, and “Blowfish” encryption algorithms, and also for the “HmacMD5” and “HmacSHA1” message authentication (MAC) algorithms.

Once you have obtained a `KeyGenerator`, you initialize it with the `init()` method. You can provide a `java.security.spec.AlgorithmParameterSpec` object to provide algorithm-specific initialization parameters or simply specify the desired size (in bits) of the key to be generated. In either case, you can also specify a source of randomness in the form of a `SecureRandom` object. If you do not specify a `SecureRandom`, the `KeyGenerator` instantiates one of its own. None of the algorithms supported by the “SunJCE” provider require algorithm-specific parameters.

After calling `getInstance()` to obtain a `KeyGenerator` and `init()` to initialize it, simply call `generateKey()` to create a new `SecretKey`. Remember that the `SecretKey` must be kept secret. Take precautions when storing or transmitting the key, so that it does not fall into the wrong hands. You may want to use a `java.security.KeyStore` object to store the key in a password-protected form.

```
public class KeyGenerator {
    // Protected Constructors
    protected KeyGenerator(KeyGeneratorSpi keyGenSpi, java.security.Provider provider, String algorithm);
    // Public Class Methods
    public static final KeyGenerator getInstance(String algorithm) throws java.security.NoSuchAlgorithmException;
    public static final KeyGenerator getInstance(String algorithm, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;
    public static final KeyGenerator getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
    // Public Instance Methods
    public final SecretKey generateKey();
    public final String getAlgorithm();
    public final java.security.Provider getProvider();
    public final void init(int keysize);
    public final void init(java.security.spec.AlgorithmParameterSpec params)
        throws java.security.InvalidAlgorithmParameterException;
    public final void init(java.security.SecureRandom random);
    public final void init(int keysize, java.security.SecureRandom random);
    public final void init(java.security.spec.AlgorithmParameterSpec params, java.security.SecureRandom random)
        throws java.security.InvalidAlgorithmParameterException;
}
```

Returned By: `KeyGenerator.getInstance()`

KeyGeneratorSpi

Java 1.4

javax.crypto

This abstract class defines the service-provider interface for `KeyGenerator`. A cryptographic provider must implement a concrete subclass of this class for each key-generation algorithm it supports. Applications never need to use or subclass this class.

KeyGeneratorSpi

```
public abstract class KeyGeneratorSpi {  
    // Public Constructors  
    public KeyGeneratorSpi();  
    // Protected Instance Methods  
    protected abstract SecretKey engineGenerateKey();  
    protected abstract void engineInit(java.security.SecureRandom random);  
    protected abstract void engineInit(int keysize, java.security.SecureRandom random);  
    protected abstract void engineInit(java.security.spec.AlgorithmParameterSpec params,  
                                         java.security.SecureRandom random)  
        throws java.security.InvalidAlgorithmParameterException;  
}
```

Passed To: KeyGenerator.KeyGenerator()

Mac

Java 1.4

javax.crypto

cloneable

This class defines an API for computing a *message authentication code* (MAC) that can check the integrity of information transmitted between two parties that share a secret key. A MAC is similar to a digital signature, except that it is generated with a secret key rather than with a public/private key pair. The **Mac** class is algorithm-independent and provider-based. Obtain a **Mac** object by calling one of the static **getInstance()** factory methods and specifying the name of the desired MAC algorithm and, optionally, the name of the provider of the desired implementation. The “SunJCE” provider implements two algorithms: “HmacMD5” and “HmacSHA1”. These are MAC algorithms based on the MD5 and SHA-1 cryptographic hash functions.

After obtaining a **Mac** object, initialize it by calling the **init()** method and specifying a **SecretKey** and, optionally, a **java.security.spec.AlgorithmParameterSpec** object. The “HmacMD5” and “HmacSHA1” algorithms can use any kind of **SecretKey**; they are not restricted to a particular cryptographic algorithm. And neither algorithm requires an **AlgorithmParameterSpec** object.

After obtaining and initializing a **Mac** object, specify the data for which the MAC is to be computed. If the data is contained in a single byte array, simply pass it to **doFinal()**. If the data is streaming or is stored in various locations, you can supply the data in multiple calls to **update()**. End the series of **update()** calls with a single call to **doFinal()**. Note that some versions of **doFinal()** return the MAC data as the function return value. Another version stores the MAC data in a byte array you supply. If you use this version of **doFinal()**, be sure to call **getMacLength()** to instantiate an array of the correct length.

A call to **doFinal()** resets the internal state of a **Mac** object. If you want to compute a MAC for part of your data and then proceed to compute the MAC for the full data, you should **clone()** the **Mac** object before calling **doFinal()**. Note, however, that **Mac** implementations are not required to implement **Cloneable**.

Object	Mac	Cloneable
--------	-----	-----------

```
public class Mac implements Cloneable {  
    // Protected Constructors  
    protected Mac(MacSpi macSpi, java.security.Provider provider, String algorithm);  
    // Public Class Methods  
    public static final Mac getInstance(String algorithm) throws java.security.NoSuchAlgorithmException;  
    public static final Mac getInstance(String algorithm, String provider)  
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;  
    public static final Mac getInstance(String algorithm, java.security.Provider provider)  
        throws java.security.NoSuchAlgorithmException;
```

```
// Public Instance Methods
public final byte[] doFinal() throws IllegalStateException;
public final byte[] doFinal(byte[] input) throws IllegalStateException;
public final void doFinal(byte[] output, int outOffset) throws ShortBufferException, IllegalStateException;
public final String getAlgorithm();
public final int getMacLength();
public final java.security.Provider getProvider();
public final void init(java.security.Key key) throws java.security.InvalidKeyException;
public final void init(java.security.Key key, java.security.spec.AlgorithmParameterSpec params)
    throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;
public final void reset();
public final void update(byte[] input) throws IllegalStateException;
public final void update(byte input) throws IllegalStateException;
public final void update(byte[] input, int offset, int len) throws IllegalStateException;
// Public Methods Overriding Object
public final Object clone() throws CloneNotSupportedException;
}
```

Returned By: Mac.getInstance()

MacSpi

Java 1.4

javax.crypto

This abstract class defines the service-provider interface for `Mac`. A cryptographic provider must implement a concrete subclass of this class for each MAC algorithm it supports. Applications never need to use or subclass this class.

```
public abstract class MacSpi {
// Public Constructors
    public MacSpi();
// Public Methods Overriding Object
    public Object clone() throws CloneNotSupportedException;
// Protected Instance Methods
    protected abstract byte[] engineDoFinal();
    protected abstract int engineGetMacLength();
    protected abstract void engineInit(java.security.Key key, java.security.spec.AlgorithmParameterSpec params)
        throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;
    protected abstract void engineReset();
    protected abstract void engineUpdate(byte input);
    protected abstract void engineUpdate(byte[] input, int offset, int len);
}
```

Passed To: Mac.Mac()

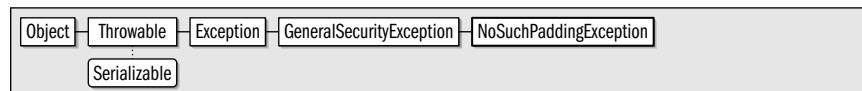
NoSuchPaddingException

Java 1.4

javax.crypto

serializable checked

This exception signals that no implementation of the requested padding scheme can be found.



```
public class NoSuchPaddingException extends java.security.GeneralSecurityException {
// Public Constructors
    public NoSuchPaddingException();
}
```

NoSuchPaddingException

```
public NoSuchPaddingException(String msg);  
}
```

Thrown By: Cipher.getInstance(), CipherSpi.engineSetPadding()

NullCipher

Java 1.4

javax.crypto

This trivial subclass of Cipher implements an identity cipher that does not transform plain text in any way. Unlike Cipher objects returned by Cipher.getInstance(), a NullCipher must be created with the NullCipher() constructor.

```
Object — Cipher — NullCipher  
  
public class NullCipher extends Cipher {  
    // Public Constructors  
    public NullCipher();  
}
```

SealedObject

Java 1.4

javax.crypto

serializable

This class is a wrapper around a serializable object. It serializes the object and encrypts the resulting data stream, thereby protecting the confidentiality of the object. Create a SealedObject by specifying the object to be sealed and a Cipher object to perform the encryption. Retrieve the sealed object by calling getObject() and specifying the Cipher or java.security.Key to use for decryption. The SealedObject keeps track of the encryption algorithm and parameters so that a Key object alone can decrypt the object.

```
Object — SealedObject — Serializable  
  
public class SealedObject implements Serializable {  
    // Public Constructors  
    public SealedObject(Serializable object, Cipher c) throws java.io.IOException, IllegalArgumentException;  
    // Protected Constructors  
    protected SealedObject(SealedObject so);  
    // Public Instance Methods  
    public final String getAlgorithm();  
    public final Object getObject(java.security.Key key) throws java.io.IOException, ClassNotFoundException,  
        java.security.NoSuchAlgorithmException, java.security.InvalidKeyException;  
    public final Object getObject(Cipher c) throws java.io.IOException, ClassNotFoundException,  
        IllegalArgumentException, BadPaddingException;  
    public final Object getObject(java.security.Key key, String provider) throws java.io.IOException,  
        ClassNotFoundException, java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException,  
        java.security.InvalidKeyException;  
    // Protected Instance Fields  
    protected byte[] encodedParams;  
}
```

Passed To: SealedObject.SealedObject()

SecretKey

Java 1.4

javax.crypto

serializable

This interface represents a secret key used for symmetric cryptographic algorithms that depend on both the sender and receiver knowing the same secret. SecretKey extends the java.security.Key interface, but does not add any new methods. The interface exists in

order to keep secret keys distinct from the public and private keys used in public-key, or asymmetric, cryptography. See also `java.security.PublicKey` and `java.security.PrivateKey`.

A secret key is nothing more than arrays of bytes and does not require a specialized encoding format. Therefore, an implementation of this interface should return the format name "RAW" from `getFormat()` and should return the bytes of the key from `getEncoded()`. (These two methods are defined by the `java.security.Key` interface that `SecretKey` extends.)

Serializable Key SecretKey

```
public interface SecretKey extends java.security.Key {
}
```

Implementations: `javax.crypto.interfaces.PBEKey`, `javax.crypto.spec.SecretKeySpec`, `javax.security.auth.kerberos.KerberosKey`

Passed To: `SecretKeyFactory.{getKeySpec(), translateKey()}`,
`SecretKeyFactorySpi.{engineGetKeySpec(), engineTranslateKey()}`

Returned By: `KeyAgreement.generateSecret()`, `KeyAgreementSpi.engineGenerateSecret()`,
`KeyGenerator.generateKey()`, `KeyGeneratorSpi.engineGenerateKey()`,
`SecretKeyFactory.{generateSecret(), translateKey()}`, `SecretKeyFactorySpi.{engineGenerateSecret(), engineTranslateKey()}`, `javax.security.auth.kerberos.KerberosTicket.getSessionKey()`

SecretKeyFactory

Java 1.4

javax.crypto

This class defines an API for translating a secret key between its opaque `SecretKey` representation and its transparent `javax.crypto.SecretKeySpec` representation. It is much like `java.security.KeyFactory`, except that it works with secret (or symmetric) keys rather than with public and private (asymmetric) keys. `SecretKeyFactory` is algorithm-independent and provider-based, so you must obtain a `SecretKeyFactory` object by calling one of the static `getInstance()` factory methods and specifying the name of the desired secret-key algorithm and, optionally, the name of the provider whose implementation is desired. The "SunJCE" provider provides `SecretKeyFactory` implementations for the "DES", "DESede", and "PBEWithMD5AndDES" algorithms.

Once you have obtained a `SecretKeyFactory`, use `generateSecret()` to create a `SecretKey` from a `java.security.spec.KeySpec` (or its subclass, `javax.crypto.spec.SecretKeySpec`) or call `getKeySpec()` to obtain a `KeySpec` for a `Key` object. Because there can be more than one suitable type of `KeySpec`, `getKeySpec()` requires a `Class` object to specify the type of the `KeySpec` to be created. See also `DESKeySpec`, `DESedeKeySpec`, and `PBEKeySpec` in the `javax.crypto.spec` package.

```
public class SecretKeyFactory {
    // Protected Constructors
    protected SecretKeyFactory(SecretKeyFactorySpi keyFacSpi, java.security.Provider provider, String algorithm);
    // Public Class Methods
    public static final SecretKeyFactory getInstance(String algorithm) throws java.security.NoSuchAlgorithmException;
    public static final SecretKeyFactory getInstance(String algorithm, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;
    public static final SecretKeyFactory getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
    // Public Instance Methods
    public final SecretKey generateSecret(java.security.spec.KeySpec keySpec)
        throws java.security.spec.InvalidKeySpecException;
```


SecretKeyFactory

```
public final String getAlgorithm();
public final java.security.spec.KeySpec getKeySpec(SecretKey key, Class keySpec)
    throws java.security.spec.InvalidKeySpecException;
public final java.security.Provider getProvider();
public final SecretKey translateKey(SecretKey key) throws java.security.InvalidKeyException;
}
```

Returned By: SecretKeyFactory.getInstance()

SecretKeyFactorySpi

Java 1.4

javax.crypto

This abstract class defines the service-provider interface for `SecretKeyFactory`. A cryptographic provider must implement a concrete subclass of this class for each type of secret key it supports. Applications never need to use or subclass this class.

```
public abstract class SecretKeyFactorySpi {
    // Public Constructors
    public SecretKeyFactorySpi();
    // Protected Instance Methods
    protected abstract SecretKey engineGenerateSecret(java.security.spec.KeySpec keySpec)
        throws java.security.spec.InvalidKeySpecException;
    protected abstract java.security.spec.KeySpec engineGetKeySpec(SecretKey key, Class keySpec)
        throws java.security.spec.InvalidKeySpecException;
    protected abstract SecretKey engineTranslateKey(SecretKey key) throws java.security.InvalidKeyException;
}
```

Passed To: SecretKeyFactory.SecretKeyFactory()

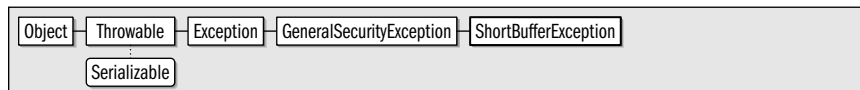
ShortBufferException

Java 1.4

javax.crypto

serializable checked

This signals that an output buffer is too short to hold the results of an operation.



```
public class ShortBufferException extends java.security.GeneralSecurityException {
    // Public Constructors
    public ShortBufferException();
    public ShortBufferException(String msg);
}
```

Thrown By: Cipher.{doFinal(), update()}, CipherSpi.{engineDoFinal(), engineUpdate()}, ExemptionMechanism.genExemptionBlob(), ExemptionMechanismSpi.engineGenExemptionBlob(), KeyAgreement.generateSecret(), KeyAgreementSpi.engineGenerateSecret(), Mac.doFinal()

Package javax.crypto.interfaces

Java 1.4

The interfaces in the `javax.crypto.interfaces` package define the public methods that must be supported by various types of encryption keys. The “DH” interfaces represent Diffie-Hellman public/private key pairs used in the Diffie-Hellman key-agreement protocol. The “PBE” interface is for Password-Based Encryption. These interfaces are typically of interest only to programmers who are implementing a cryptographic provider or who

want to implement cryptographic algorithms themselves. Use of this package requires basic familiarity with the encryption algorithms and the mathematics that underlie them. Note that the `javax.crypto.spec` package contains classes that provide algorithm-specific details about encryption keys.

Interfaces:

public interface **DHKey**;
public interface DHPrivateKey extends **DHKey**, **java.security.PrivateKey**;
public interface DHPublicKey extends **DHKey**, **java.security.PublicKey**;
public interface PBEKey extends **javax.crypto.SecretKey**;

DHKey

Java 1.4

`javax.crypto.interfaces`

This interface represents a Diffie-Hellman key. The `javax.crypto.spec.DHParameterSpec` returned by `getParams()` specifies the parameters that generate the key; they define a key family. See the subinterfaces `DHPublicKey` and `DHPrivateKey` for the actual key values.

```
public interface DHKey {
    // Public Instance Methods
    public abstract javax.crypto.spec.DHParameterSpec getParams();
}
```

Implementations: `DHPrivateKey`, `DHPublicKey`

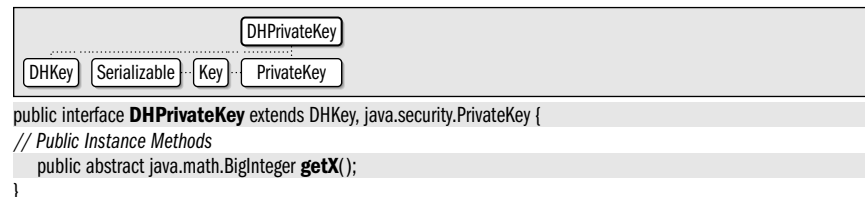
DHPrivateKey

Java 1.4

`javax.crypto.interfaces`

serializable

This interface represents a Diffie-Hellman private key. Note that it extends two interfaces: `DHKey` and `java.security.PrivateKey`. `getX()` returns the private-key value. If you are working with a `PrivateKey` you know is a Diffie-Hellman key, you can cast your `PrivateKey` to a `DHPrivateKey`.



DHPublicKey

Java 1.4

`javax.crypto.interfaces`

serializable

This interface represents a Diffie-Hellman public key. Note that it extends two interfaces: `DHKey` and `java.security.PublicKey`. `getY()` returns the public-key value. If you are working with a `PublicKey` you know is a Diffie-Hellman key, you can cast your `PublicKey` to a `DHPublicKey`.



DHPublicKey

```
public abstract java.math.BigInteger getY();  
}
```

PBEKey

Java 1.4

javax.crypto.interfaces

serializable

This interface represents a key for password-based encryption. If you are working with a `SecretKey` that you know is a password-based key, you can cast it to a `PBEKey`.

Serializable Key SecretKey PBEKey

```
public interface PBEKey extends javax.crypto.SecretKey {  
    // Public Instance Methods  
    public abstract int getIterationCount();  
    public abstract char[] getPassword();  
    public abstract byte[] getSalt();  
}
```

Package javax.crypto.spec

Java 1.4

The `javax.crypto.spec` package contains classes that define transparent `java.security.spec.KeySpec` and `java.security.spec.AlgorithmParameterSpec` representations of secret keys, Diffie-Hellman public and private keys, and parameters used by various cryptographic algorithms. The classes in this package are used in conjunction with `java.security.KeyFactory`, `javax.crypto.SecretKeyFactory` and `java.security.AlgorithmParameters` for converting opaque `Key`, and `AlgorithmParameters` objects to and from transparent representations. In order to make good use of this package, you must be familiar with the specifications of the various cryptographic algorithms it supports and the basic mathematics that underlie those algorithms.

Classes:

```
public class DESedeKeySpec implements java.security.spec.KeySpec;  
public class DESKeySpec implements java.security.spec.KeySpec;  
public class DHGenParameterSpec implements java.security.spec.AlgorithmParameterSpec;  
public class DHParameterSpec implements java.security.spec.AlgorithmParameterSpec;  
public class DHPrivateKeySpec implements java.security.spec.KeySpec;  
public class DHPublicKeySpec implements java.security.spec.KeySpec;  
public class IvParameterSpec implements java.security.spec.AlgorithmParameterSpec;  
public class PBEKeySpec implements java.security.spec.KeySpec;  
public class PBEPParameterSpec implements java.security.spec.AlgorithmParameterSpec;  
public class RC2ParameterSpec implements java.security.spec.AlgorithmParameterSpec;  
public class RC5ParameterSpec implements java.security.spec.AlgorithmParameterSpec;  
public class SecretKeySpec implements java.security.spec.KeySpec, javax.crypto.SecretKey;
```

DESedeKeySpec

Java 1.4

javax.crypto.spec

This class is a transparent representation of a DESede (triple-DES) key. The key is 24 bytes long.

Object DESedeKeySpec KeySpec

```

public class DESedeKeySpec implements java.security.spec.KeySpec {
    // Public Constructors
    public DESedeKeySpec(byte[] key) throws java.security.InvalidKeyException;
    public DESedeKeySpec(byte[] key, int offset) throws java.security.InvalidKeyException;
    // Public Constants
    public static final int DES_EDE_KEY_LEN; =24
    // Public Class Methods
    public static boolean isParityAdjusted(byte[] key, int offset) throws java.security.InvalidKeyException;
    // Public Instance Methods
    public byte[] getKey();
}

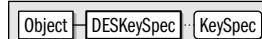
```

DESKeySpec

Java 1.4

javax.crypto.spec

This class is a transparent representation of a DES key. The key is 8 bytes long.



```

public class DESKeySpec implements java.security.spec.KeySpec {
    // Public Constructors
    public DESKeySpec(byte[] key) throws java.security.InvalidKeyException;
    public DESKeySpec(byte[] key, int offset) throws java.security.InvalidKeyException;
    // Public Constants
    public static final int DES_KEY_LEN; =8
    // Public Class Methods
    public static boolean isParityAdjusted(byte[] key, int offset) throws java.security.InvalidKeyException;
    public static boolean isWeak(byte[] key, int offset) throws java.security.InvalidKeyException;
    // Public Instance Methods
    public byte[] getKey();
}

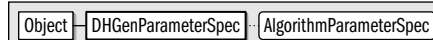
```

DHGenParameterSpec

Java 1.4

javax.crypto.spec

This class is a transparent representation of the values needed to generate a set of Diffie-Hellman parameters (see `DHParameterSpec`). An instance of this class can be passed to the `init()` method of a `java.security.AlgorithmParameterGenerator` that computes Diffie-Hellman parameters.



```

public class DHGenParameterSpec implements java.security.spec.AlgorithmParameterSpec {
    // Public Constructors
    public DHGenParameterSpec(int primeSize, int exponentSize);
    // Public Instance Methods
    public int getExponentSize();
    public int getPrimeSize();
}

```

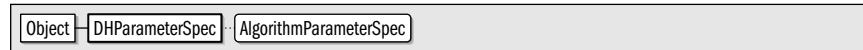
DHParameterSpec

DHParameterSpec

Java 1.4

javax.crypto.spec

This class is a transparent representation of the set of parameters required by the Diffie-Hellman key-agreement algorithm. All parties to the key agreement must share these parameters and use them to generate a Diffie-Hellman public/private key pair.



```
public class DHParameterSpec implements java.security.spec.AlgorithmParameterSpec {  
    // Public Constructors  
    public DHParameterSpec(java.math.BigInteger p, java.math.BigInteger g);  
    public DHParameterSpec(java.math.BigInteger p, java.math.BigInteger g, int l);  
    // Public Instance Methods  
    public java.math.BigInteger getG();  
    public int getL();  
    public java.math.BigInteger getP();  
}
```

Returned By: javax.crypto.interfaces.DHKey.getParams()

DHPrivateKeySpec

Java 1.4

javax.crypto.spec

This java.security.spec.KeySpec is a transparent representation of a Diffie-Hellman private key.



```
public class DHPrivateKeySpec implements java.security.spec.KeySpec {  
    // Public Constructors  
    public DHPrivateKeySpec(java.math.BigInteger x, java.math.BigInteger p, java.math.BigInteger g);  
    // Public Instance Methods  
    public java.math.BigInteger getG();  
    public java.math.BigInteger getP();  
    public java.math.BigInteger getX();  
}
```

DHPublicKeySpec

Java 1.4

javax.crypto.spec

This java.security.spec.KeySpec is a transparent representation of a Diffie-Hellman public key.



```
public class DHPublicKeySpec implements java.security.spec.KeySpec {  
    // Public Constructors  
    public DHPublicKeySpec(java.math.BigInteger y, java.math.BigInteger p, java.math.BigInteger g);  
    // Public Instance Methods  
    public java.math.BigInteger getG();  
    public java.math.BigInteger getP();  
    public java.math.BigInteger getY();  
}
```

IvParameterSpec

Java 1.4

javax.crypto.spec

This `java.security.spec.AlgorithmParameterSpec` is a transparent representation of an initialization vector, or IV. An IV is required for block ciphers used in feedback mode, such as DES in CBC mode.



```

public class IvParameterSpec implements java.security.spec.AlgorithmParameterSpec {
    // Public Constructors
    public IvParameterSpec(byte[] iv);
    public IvParameterSpec(byte[] iv, int offset, int len);
    // Public Instance Methods
    public byte[] getIV();
}

```

PBEKeySpec

Java 1.4

javax.crypto.spec

This class is a transparent representation of a password used in password-based encryption (PBE). The password is stored as a `char` array rather than as a `String` so that the characters of the password can be overwritten when they are no longer needed (for increased security).



```

public class PBEKeySpec implements java.security.spec.KeySpec {
    // Public Constructors
    public PBEKeySpec(char[] password);
    public PBEKeySpec(char[] password, byte[] salt, int iterationCount);
    public PBEKeySpec(char[] password, byte[] salt, int iterationCount, int keyLength);
    // Public Instance Methods
    public final void clearPassword();
    public final int getIterationCount();
    public final int getKeyLength();
    public final char[] getPassword();
    public final byte[] getSalt();
}

```

PBEPParameterSpec

Java 1.4

javax.crypto.spec

This class is a transparent representation of the parameters used with the password-based encryption algorithm defined by PKCS#5.



```

public class PBEPParameterSpec implements java.security.spec.AlgorithmParameterSpec {
    // Public Constructors
    public PBEPParameterSpec(byte[] salt, int iterationCount);
    // Public Instance Methods
    public int getIterationCount();
    public byte[] getSalt();
}

```

RC2ParameterSpec

Java 1.4

javax.crypto.spec

This class is a transparent representation of the parameters used by the RC2 encryption algorithm. An object of this class initializes a `Cipher` object that implements RC2. Note that the “SunJCE” provider supplied by Sun does not implement RC2.



```

public class RC2ParameterSpec implements java.security.spec.AlgorithmParameterSpec {
    // Public Constructors
    public RC2ParameterSpec(int effectiveKeyBits);
    public RC2ParameterSpec(int effectiveKeyBits, byte[] iv);
    public RC2ParameterSpec(int effectiveKeyBits, byte[] iv, int offset);
    // Public Instance Methods
    public int getEffectiveKeyBits();
    public byte[] getIV();
    // Public Methods Overriding Object
    public boolean equals(Object obj);
    public int hashCode();
}

```

RC5ParameterSpec

Java 1.4

javax.crypto.spec

This class is a transparent representation of the parameters used by the RC5 encryption algorithm. An object of this class initializes a `Cipher` object that implements RC5. Note that the “SunJCE” provider supplied by Sun does not implement RC5.



```

public class RC5ParameterSpec implements java.security.spec.AlgorithmParameterSpec {
    // Public Constructors
    public RC5ParameterSpec(int version, int rounds, int wordSize);
    public RC5ParameterSpec(int version, int rounds, int wordSize, byte[] iv);
    public RC5ParameterSpec(int version, int rounds, int wordSize, byte[] iv, int offset);
    // Public Instance Methods
    public byte[] getIV();
    public int getRounds();
    public int getVersion();
    public int getWordSize();
    // Public Methods Overriding Object
    public boolean equals(Object obj);
    public int hashCode();
}

```

SecretKeySpec

Java 1.4


javax.crypto.spec

serializable

This class is a transparent and algorithm-independent representation of a secret key. This class is useful only for encryption algorithms (such as DES and DESede) whose secret keys can be represented as arbitrary byte arrays and do not require auxiliary

SecretKeySpec

parameters. Note that `SecretKeySpec` implements the `javax.crypto.SecretKey` interface directly, so no algorithm-specific `javax.crypto.SecretKeyFactory` object is required.



```
public class SecretKeySpec implements java.security.spec.KeySpec, javax.crypto.SecretKey {  
    // Public Constructors  
    public SecretKeySpec(byte[] key, String algorithm);  
    public SecretKeySpec(byte[] key, int offset, int len, String algorithm);  
    // Methods Implementing Key  
    public String getAlgorithm();  
    public byte[] getEncoded();  
    public String getFormat();  
    // Public Methods Overriding Object  
    public boolean equals(Object obj);  
    public int hashCode();  
}
```