# DATABASE PROGRAMMING WITH JAVA

## Session: 4

# Cryptography

# Objectives

- Explain Cryptography
- Describe Cryptography Architecture
- Explain Java Cryptography Extension
- Describe the Cipher class
- Explain password-based encryption
- Explain the new security features in Java

# Introduction to Cryptography

Cryptography is a study of techniques used for secret and secure communication in the presence of adversaries.

## Encryption and Decryption

- Encryption converts plain text or original data into an encrypted form or the ciphertext by applying mathematical transformations.
- Decryption retrieves the original data or the plaintext from the ciphertext using a decryption key. It can be termed as reverse process of encryption.

# Definition of Cryptography

Cryptography is the mechanism of encoding information in a secret coded form, intended only for the recipient to access the information.

Cryptography is used to keep communications limited and private to only the sender and receiver.

Enhances the security, authenticity, and integrity of the information passed across the communication medium.

# Cryptography Schemes

## Secret key cryptography

- This is also known as Symmetric cryptography, as the two entities taking part in the communication process, must share the same secret key.

## Public key cryptography

- Operates with two different keys, one is used for encoding and the second is used for decoding the data.

## Hash functions

- Makes use of a mathematical hash function to encrypt the information into an irreversible code.

# Need for Cryptography

Cryptography is needed to identify all the possible threats to the application and provide measures to prevent them. The threats are categorized as follows:

**Violation of privacy and confidentiality**

- Data can be read by an unintended receiver, when it is sent via a network.

**Tampering**

- Tampering means modifying or deleting a resource without proper access privilege.

**Spoofing**

- Also known as identity interception, which means impersonating the identity of a different user and use it in an unauthorized way.

# Purpose of Cryptography

Cryptographic techniques are used to serve the following purposes:

**Authentication**

- To provides a way of verifying a users identity, to prevent spoofing.

**Privacy and Confidentiality**

- To prevent any unintended receiver from reading the data.
- To achieve data confidentiality, the data can be encrypted.

**Integrity**

- To protect data from getting tampered, while it is on the network.

**Non Repudiation**

- To ensure that a user or a business organization or a program entity has performed a transaction.

# Key Agreement

Both the communicating entities should agree on the key used for encryption and decryption. Java supports key agreement operations by providing the `javax.crypto.KeyAgreement` class and its related methods.

The steps for using the key agreement are as follows:

1. The sender generates a public and private key pair and sends the public key, along with the algorithm specification that was agreed with the receiver. Diffie-Hellamn algorithm is one such algorithm.
2. The receiver generates own public and private key pair using the same algorithm and sends the public key to the sender.
3. The sender generates the secret key using own private key and the receivers public key.
4. The receiver generates the same secret key using own private key and the senders public key using Diffie-Hellamn algorithm. The algorithm ensures that both parties generate the same secret key.

# Java Cryptography Architecture (JCA)

Java Cryptography Architecture (JCA) is a framework written in Java to access and develop cryptographic functionality and forms part of the Java security API.

Java Cryptography Architecture defines various classes implementing various cryptographic algorithms.

These classes are in turn used to generate message digests, public-private key pairs, and so on.

Java Cryptography Extension (JCE) extends the underlying architecture of JCA framework to implement encryption, key exchange.

JCA and JCE together provide a complete, platform-independent API to implement cryptography.

JCA also specifies the design patterns and architecture to define cryptographic concepts and algorithms.

# Design Principles of JCA

The JCA was designed to access cryptography and security concepts. The basic design principles on which the Java Cryptography Architecture (JCA) was founded are as follows:

## Implementation independence and interoperability

- Aims at utilizing the cryptographic concepts, without being concerned about the implementations, which is achieved through cryptography service provider.
- Different implementations can work with each other.

## Algorithm independence and extensibility

- Ensures the implementation of cryptographic algorithms.
- Makes sure that new algorithms that correspond to one of the supported service provider classes can be added easily.

# Components of JCA Architecture 1-2

Java Cryptography Architecture (JCA) defines the following components:

## Cryptographic Service Providers

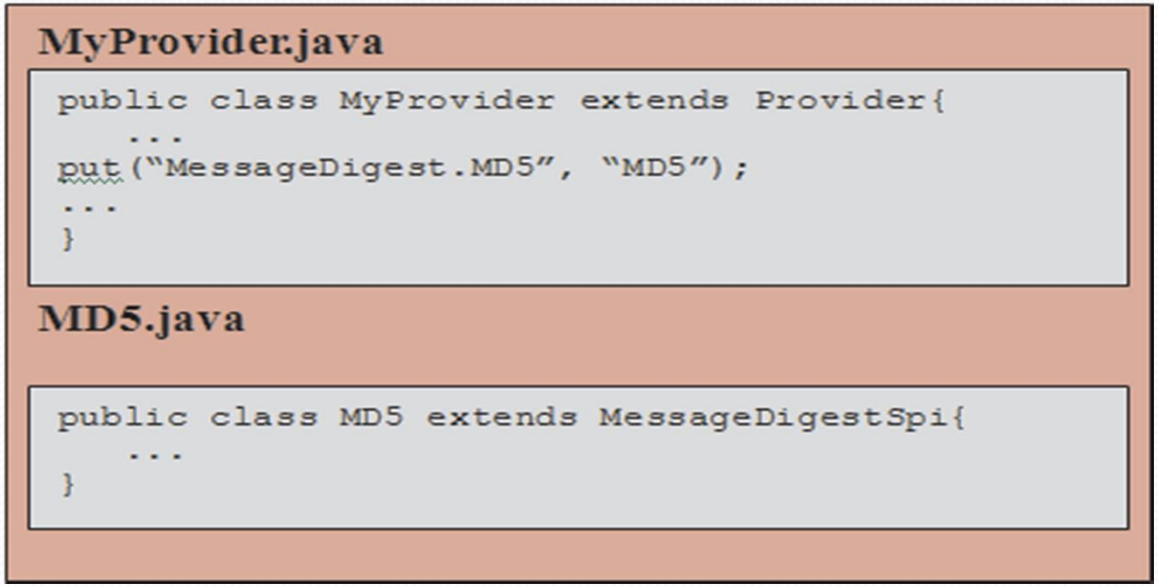
- Any application which requires cryptographic services has to obtain them from the `Provider` class. This `Provider` class implements services such as:
  - Cryptographic algorithms such as `DES`, `RSA`, `DSA`, and `MD-5`.
  - Key generation, management facilities.

## Key Management

- The JCA also defines a database called keystore to manage the library of keys and certificates.

# Components of JCA Architecture 2-2

Following figure shows the implementation of a `Provider` class:




The implementations provided by the `Provider` class are termed as Engine classes.

There are various Engine classes such as `MessageDigest`, `SecretKeyFactory`, `CertificateFactory`, `KeyStore`, and so on.

# Cryptographic Service

- The Service provider classes are also known as Engine classes that provide the functionality of a type of cryptographic algorithm.
- A service of an Engine class can be requested as per the following syntax:

**Syntax:**

```
static EngineClassName getInstance(String algorithm)
throws NoSuchAlgorithmException
static EngineClassName getInstance(String algorithm, String
provider)throws NoSuchAlgorithmException,
NoSuchProviderException
static EngineClassName getInstance(String algorithm,
Provider provider)throws NoSuchAlgorithmException
```

- Following code shows an example of Engine class:
  - `KeyPairGenerator kp = KeyPairGenerator.getInstance("DHA");`

# Classes and Interfaces of JCA 1-2

Following table shows the Core Engine classes of JCA:

| Class | Description |
|---|---|
| `MessageDigest` | The class is used to calculate the message digest or hash value of the specified data. |
| `Signature` | This class is initialized with keys and is used in digitally sign the documents and verify the digital signatures. |
| `KeyPairGenerator` | The class is used to produce a pair of public and private keys appropriate for a specified algorithm. |
| `KeyFactory` | The class is used to transform opaque keys of type Key into key specifications and provide transparent representations of the underlying key material and vice versa. |
| `CertificateFactory` | The class is used to generate public key certificates and Certificate Revocation Lists. |
| `Message Authentication Codes (MAC)` | This class is used to generate the message authentication code or message digest for a message. |
| `SecretKeyFactory` | This class is used for generating symmetric keys. |
| `Cipher` | This class has to be initialized with keys, it is used for encrypting/decrypting data using various algorithms. |

# Classes and Interfaces of JCA 2-2

| Class | Description |
|---|---|
| SecureRandom | The class is used to generate random and pseudorandom numbers. |
| KeyGenerator | This class is used to generate keys specific to an algorithm. |
| KeyAgreement | This class is used to establish the cryptographic key for certain communication session. |
| AlgorithmParameters | This class is used to store the parameters for a specific algorithm. |
| Algorithm ParameterGenerator | This class is used to generate the required parameters for a specific algorithm. |
| CertPathBuilder | This class is used to build certificate chains. |
| CertPathValidator | This class is used to certificate chains. |
| CertStore | This class is used to retrieve certificates and Certificate Revocation Lists. |
| Key | The interface is used as the top-level interface for all opaque keys. It specifies the functionality shared by all opaque key objects. |
| KeySpec | The interface is used to group and provide type safety for all key specifications without any methods or constants. This interface must be implemented by all key specifications. |

# Overview of Java Cryptography Extension (JCE)

Java Cryptography Extension (JCE) is a set of packages that form a framework and provides implementations for encryption, key generation and agreement, and Message Authentication Code (MAC) algorithms. The JCE API provides the support for following:

- Symmetric Block Encryption
- Symmetric Stream Encryption
- Password Based Encryption
- Key Agreement
- Message authentication

# Packages in JCE

Following table shows packages in JCE:

| Class | Description |
|---|---|
| `java.security` | The package provides classes and interfaces to form the security framework. |
| `java.security.spec` | The package provides classes and interfaces for key and algorithm parameter specifications. |
| `javax.crypto` | The package provides classes and interfaces to perform cryptographic operations. |
| `javax.crypto.spec` | The package provides classes and interfaces for key specifications and algorithm parameter specifications. |
| `java.security.cert` | The package provides classes and interfaces for parsing and managing certificates, certificate revocation lists, and certification paths. |
| `java.security.interfaces` | The package provides interfaces for generating RSA keys. |
| `java.crypto.interfaces` | The package provides interfaces for Diffie-Hellman keys. |

Encryption in JCA is implemented through the `Cipher` class.

## Introduction to `Ciphers`

- Cipher is the object capable of performing encryption and decryption as per an encryption algorithm.
- This class belongs to the `javax.crypto` package and forms the base of the Java Cryptographic Extension (JCE) framework.

## `Cipher` class

- The `Cipher` class provides a set of methods to accomplish the encryption and decryption.

# Cipher Class 1-4

Following are the methods provided by `Cipher` class:

- **`getInstance()`**

  The static method returns a `Cipher` object to implement the transformation specified by the provider. Following is the syntax of the `getInstance()` method:

  **Syntax**:

```
public static final Cipher getInstance(String transformation,
String provider) throws NoSuchAlgorithmException,
NoSuchProviderException, NoSuchPaddingException
```

- **`init()`**

  The method initializes the Cipher for a specific mode of operation and also describes a key for the Cipher, which can be `ENCRYPT_MODE, DECRYPT_MODE, WRAP_MODE, or UNWRAP_MODE.`

# Cipher Class 2-4

- Following is the syntax of `init()` method:

**Syntax**:

```
public final void init(int opmode, Key key) throws
InvalidKeyException
```

- **`update()`**

  The method performs encryption or decryption on multiple parts of input data and generates partial output data. Following is the syntax of the `update()` method:

**Syntax**:

```
public final byte[] update(byte[] input) throws
IllegalStateException
```

# Cipher Class 3-4

- **`doFinal()`**

  The method feeds the last part of the input data to the `Cipher` object and produces the last part of the output data. The syntax is as follows:

**Syntax**:

```
public final byte[] doFinal() throws IllegalStateException,
IllegalBlockSizeException, BadPaddingException
```

- **`getBlockSize()`**

  The method returns the size of the block in bytes, if the `Cipher` object represents a block cipher. The syntax is as follows:

**Syntax**:

```
public final int getBlockSize()
```

# Cipher Class 4-4

- **`getAlgorithm()`**

  The method returns the name of the algorithm for the `Cipher` object. The syntax is as follows:

  **Syntax**:

  ```
  public final String getAlgorithm()
  ```

- **`getProvider()`**

  The method returns the name of the provider of this `Cipher` object. The syntax is as follows:

  **Syntax**:

  ```
  public final Provider getProvider()
  ```

# Steps for Encryption 1-3

- **Creating a `Cipher` object**

  For all engine classes, a `Cipher` object is created by invoking one of the static `getInstance()` factory methods on the `Cipher` class.

  The following Code Snippet shows the usage of `Cipher` object:

**Code Snippet**:

```
//For SunJCE provider
Cipher c = Cipher.getInstance("DES");
```

- **Initializing a `Cipher` object**

  The `Cipher` object, instantiated by the `getInstance()` method needs to be initialized with the help of `init()` method. The following Code Snippet shows the initialization:

**Code Snippet**:

```
//initialize cipher for encryption
c.init(Cipher.ENCRYPT_MODE, myKey);
```

# Steps for Encryption 2-3

- **Encrypting and Decrypting data**

  Encryption or decryption can be performed on the input data in a single step and single part operation, or in multiple steps and multiple part operations.

  The following Code Snippet demonstrates decryption in multiple parts:

**Code Snippet**:

```
//encrypt data and store in cipherText
byte[] cipherText = c.doFinal("Password". getBytes());
//initialise cipher for decryption with same key
c.init(Cipher.DECRYPT_MODE, myKey);
//decrypt data and store in plainText
byte[] plainText = cipher.doFinal(cipherText);
```

# Steps for Encryption 3-3

- **Wrapping a key**

  Wrapping of a key ensures the secure transfer of keys from one place to another.

  The following Code Snippet shows the method of wrapping and unwrapping a key:

**Code Snippet**:

```
//Wrapping a key
c.init(Cipher.WRAP_MODE,key);
byte[] wrappedKey = cipher.wrap(key);
//unwrapping a key
c.init(Cipher.UNWRAP_MODE, wrappedKey,
wrapKeyAlgParameters, random);
Key unwrappedKey=c.unwrap(wrappedKey, wrapKeyAlg,
Cipher.SECRET_KEY);
```

# Cipher Block

You can encrypt single bits or a block of bits called ‘cipher blocks’.

Block cipher algorithms like BlowFish or DES requires the input to be an exact multiple of the block size.

A short block is padded either with all zeros or ones.

One of the most common padding techniques used is PKCS6.

With PKCS5 padding technique, a short block is padded with a repeating byte. The value of the byte is determined by the number of remaining bytes.

# Cipher Objects 1-9

- Cipher objects are created using the `getInstance()` method of the `Cipher` class. It implements a specified transformation on the input data to produce output.
- The transformation can have one of the following forms:
  - "algorithm/mode/padding", such as "`DES/CBC/PKCS5Padding`"
  - "(only) algorithm", such as "`DES`"
- Following syntax shows the usage of Cipher objects through `getInstance()` method:

**Syntax**:

```
public static Cipher getInstance(String transformation)
public static Cipher getInstance(String transformation, String
provider)
```

- Examples for transformation are `DES/CFB8/NoPadding` and `DES/OFB32/PKCS5Padding` transformations.

# Cipher Objects 2-9

- The `Cipher` object is initialized by the `init()` method.
- Following is the syntax for using `init()` method:

**Syntax**:

```
public void init(int opmode, Key key)
```

- `Opmode` can have any of the following values:
  - `ENCRYPT_MODE` - for encryption of data
  - `DECRYPT_MODE` - for decryption of data
  - `WRAP_MODE` - for wrapping a Key into bytes, so that the key can be securely transported
  - `UNWRAP_MODE` - for unwrapping of a previously wrapped key into a `java.security.Key` object

# Cipher Objects 3-9

- The following Code Snippet shows encryption and decryption process:

**Code Snippet**:

```
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.KeyGenerator;
import java.security.AlgorithmParameters;
import java.security.InvalidAlgorithmParameterException;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;
import java.security.spec.InvalidKeySpecException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import java.io.UnsupportedEncodingException;
import java.io.IOException;
public class EncrypterDecrypter {
Cipher ecipher;
Cipher dcipher;
```

# Cipher Objects 4-9

```
EncrypterDecrypter(SecretKey key, String algorithm) {
   try{
   ecipher = Cipher.getInstance(algorithm);
   dcipher = Cipher.getInstance(algorithm);
   ecipher.init(Cipher.ENCRYPT_MODE, key);
   dcipher.init(Cipher.DECRYPT_MODE, key);
   } catch (NoSuchAlgorithmException e) {
   }
   catch (InvalidKeyException e) {
   }
   catch (NoSuchPaddingException e) {
   }
  }
}
// encrypt() inputs a string and returns an encrypted version
// of that String.
public String encrypt(String str) {
try {
// Encode the string into bytes using utf-8
byte[] utf8 = str.getBytes("UTF8");
```

# Cipher Objects 5-9

```
// Encrypt
byte[] enc = ecipher.doFinal(utf8);
// Encode bytes to base64 to get a string
return new sun.misc.BASE64Encoder().encode(enc);
}catch (BadPaddingException e) {
} catch (IllegalBlockSizeException e) {
} catch (UnsupportedEncodingException e) {
} catch (IOException e) {
}
return null;
}

// decrypt() inputs a string and returns an encrypted version
// of that String.
public String decrypt(String str) {
try {
// Decode base64 to get bytes
byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(str);
// Decrypt
byte[] utf8 = dcipher.doFinal(dec);
```

# Cipher Objects 6-9

```
// Decode using utf-8
return new String(utf8, "UTF8");
} catch (BadPaddingException e) {
} catch (IllegalBlockSizeException e) {
} catch (UnsupportedEncodingException e) {
} catch (IOException e) {
}
return null;
}
// The following method is used for encrypting and decrypting
// String using symmetric Secret Key using "DES" algorithm
public static void useSecretKey() {
try {
String stringToBeEncrypted = "String to be encrypted";
SecretKey desKey = KeyGenerator.getInstance("DES").
generateKey();
// Create encrypter/decrypter class
EncrypterDecrypter encrypter = new EncrypterDecrypter(desKey,
desKey.getAlgorithm());
// Encrypt the string
String encryptedString = encrypter.encrypt(stringToBeEncrypted);
```

# Cipher Objects 7-9

```
// Decrypt the string
String decrypterString = encrypter.decrypt(encryptedString);

// Display values
System.out.println("\tEncryption algorithm Name:" +
desKey.getAlgorithm());
System.out.println("\tOriginal String : " + stringToBeEncrypted);
System.out.println("\tEncrypted String : " + encryptedString);
}
```

- The parameters used by the cipher implementation can be retrieved from the Cipher object by calling its `getParameters()` method.
- Following is the syntax of using `getParameters()` method:

**Syntax**:

```
public final AlgorithmParameters getParameters()
```

# Cipher Objects 8-9

- The following Code Snippet demonstrates how to retrieve algorithm parameters:

**Code Snippet**:

```
...
SecretKey myKey =
KeyGenerator.getInstance("DES").generateKey();
// get cipher object for password-based encryption
Cipher cipher = Cipher.getInstance("PBEWithMD5AndDES");
// initialize cipher for encryption, without supplying
// any parameters.
cipher.init(Cipher.ENCRYPT_MODE, myKey);
// encrypt some data and store ciphertext
byte[] cipherText = cipher.doFinal("text".getBytes());
// retrieve parameters generated by underlying cipher
implementation
...
```

# Cipher Objects 9-9

```
AlgorithmParameters algoParams = cipher.getParameters();
// retrieve provider information for this implementation
cipher.getProvider().getInfo();
// get parameter encoding and store it away
byte[] encodedAlgParams = algoParams.getEncoded();
...
```

- Following method in `Cipher` class can be used to determine the size of output buffer:

**Syntax**:

```
public final int getOutputSize(int length)
```

# Password-Based Encryption

Password-based encryption uses the password of the user to generate keys used in encryption.

Java has classes such as `PBEKeySpec`, `PBEParameterSpec`, and `SecretKeyFactory` that have methods required for Password-Based Encryption.

Password-Based Encryption (PBE) generates a secret encryption key based on a password provided by the end user.

PBE is specifically used in systems such as local file encryption tools, which ensures the data confidentiality.

**Explaining 'Salt' in PBE**

PBE algorithm often mix in a random number with the password, called the salt.

Performing the digest operation with a salt, it becomes difficult for the hacker to calculate the password for every value of the salt.

# PBEKeySpec Class

- The `PBEKeySpec` class is used in a Password-Based Encryption.
- Following is the syntax of using `PBEKeySpec` class to generate key based on password:

**Syntax**:

```
public class javax.crypto.spec.PBEKeySpec
extends java.lang.Object
implements java.security.spec.KeySpec
{
//Constructors
public PBEKeySpec(char[] password);
//Instance Methods
public final char[] getPassword();
}
```

- The following Code Snippet shows usage of `PBEKeySpec` object:

**Code Snippet**:

```
char[] password = "sasquatch".toCharArray();
PBEKeySpec keySpec = new PBEKeySpec(password);
```

# PBEParameterSpec Class 1-2

- The `PBEParameterSpec` class denotes the set of parameters used with Password-Based Encryption (PBE), as defined in the PKCS #5 standard.
- At the time of creating a cipher object, the salt and the iteration count is passed to make use of the PBE. The syntax is as follows:

**Syntax**:

```
public class javax.crypto.spec.PBEParameterSpec
extends java.lang.Object
implements java.security.spec.AlgorithmParameterSpec
{
//Constructors
public PBEParameterSpec(byte[] salt, int iterationCount);
//Instance Methods
public int getIterationCount(); //returns iteration count
public byte[] getSalt();//returns the salt
}
```

# PBEParameterSpec Class 2-2

- The following Code Snippet creates a `Cipher` object using the `PBEParameterSpec` class:

**Code Snippet**:

```
// Create PBE parameter set
PBEParameterSpec pSpec = new PBEParameterSpec(salt,
noOfIterations);

//Create PBE Cipher
Cipher cipher = Cipher.getInstance("PBEWithSHAAndTwofish-
CBC");

//Initialise PBE Cipher with key and parameters
cipher.init(Cipher.ENCRYPT_MODE, theKey, paramSpec);
```

# SecretKeyFactory Class 1-2

- The `SecretKeyFactory` class from `javax.crypto` package is used to build an opaque cryptographic key object from a user specified key material.
- To convert the password into a PBE key, an instance of `SecretKeyFactory` class is used.
- The `SecretKeyFactory` instance calls the `generateSecret()` method to create the secret key for the given key specification.
- The following Code Snippet shows the usage of `SecretKeyFactory` class:

**Code Snippet**:

```
SecretKeyFactory factory =
SecretKeyFactory.getInstance("PBEWithMD5AndDES") ;
SecretKey theKey = factory.generateSecret(k, eySpec);
```

# SecretKeyFactory Class 2-2

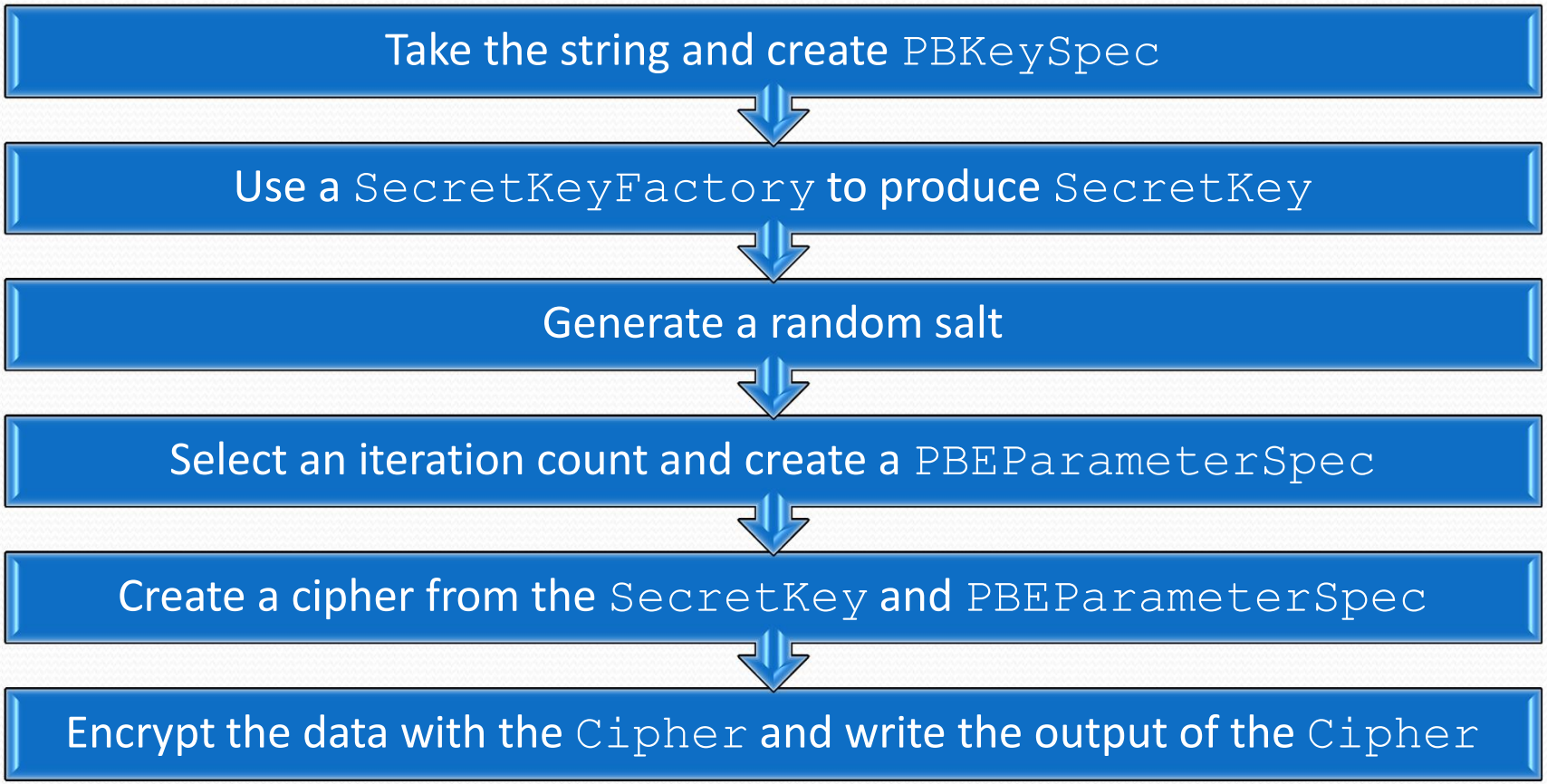
- Following is the syntax for using `SecretKeyFactory` interface:

**Syntax**:

```
public class javax.crypto.SecretKeyFactory extends
java.lang.Object
{
//Constructor
protected SecretKeyFactory(SecretKeyFactorySpi keyFactSpi,
Provider provider, String algorithm)
//Class Methods
public static final SecretKeyFactory getInstance(String);
public static final SecretKeyFactory getInstance(String,
String);
//Instance Methods
public final SecretKey generateSecret(KeySpec);
public final KeySpec getKeySpec(SecretKey, Class);
public final Provider getProvider();
public final SecretKey translateKey(SecretKey);
}
```

# Encrypting Data Using Passwords 1-5

- Following are the steps to encrypt data using the Password-Based Encryption:



# Encrypting Data Using Passwords 2-5

- **Take the string and create `PBKeySpec`**

  In this first step, a user-chosen password is entered as a string and is converted to a character array and stored by an instance of `PBEKeySpec` class. The following Code Snippet demonstrates this:

**Code Snippet**:

```
PBEKeySpec pbeks = new
PBEKeySpec(password.toCharArray());
```

- **Use a `SecretKeyFactory` to produce `SecretKey`**

  In this step, an instance of `SecretKeyFactory` class is created by providing an algorithm name based on which the key will be generated. The following Code Snippet demonstrates this step:

**Code Snippet**:

```
SecretKeyFactory skf1 =
SecretKeyFactory.getInstance("PBEWithMD5AndTri pleDES");
SecretKey key = skf1.generateSecret(pbeks);
```

# Encrypting Data Using Passwords 3-5

- **Generate random salt**

  A salt is generated randomly by creating an instance of `SecureRandom` class. The following Code Snippet demonstrates this step:

**Code Snippet**:

```
byte[] salt = new byte[8];
SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
sr.nextBytes(salt);
```

- **Select an iteration count and create a `PBEParameterSpec`**

  A constant integer value is selected as an iteration count, which is passed along with random salt to encryption process. The following Code Snippet demonstrates the usage:

**Code Snippet**:

```
final int iterationCount = 200;
PBEParameterSpec pbeps = new PBEParameterSpec(salt, 200);
```

# Encrypting Data Using Passwords 4-5

- **Create a cipher from the `SecretKey` and `PBEParameterSpec`**

  A cipher object must be created by using the `getInstance()` method with the operational mode initialized. The following Code Snippet demonstrates the creation of cipher:

**Code Snippet**:

```
Cipher cip =
Cipher.getInstance("PBEWithMD5AndTripleDES/CBC/
PKCS5Padding");
cip.init(Cipher.ENCRYPT_MODE, key, pbeps);
```

- **Encrypt the data with the Cipher and write the output of the Cipher**

  The input data is encrypted with the help of `update()` and `doFinal()` methods.

# Encrypting Data Using Passwords 5-5

The following Code Snippet shows encryption of data:

**Code Snippet**:

```
FileInputStream fileIn = new FileInputStream(args[0]);
FileOutputStream fileOut = new FileOutputStream(args[1]);
fileOut.write(salt);
byte[] buffer = new byte[4096];
int i;
while ((i = fIn.read(buffer)) != -1)
{
fOut.write(c.update(buffer, 0, i));
}
fOut.write(c.doFinal());
fOut.close();
fIn.close();
```

# New Security Features in Java

- Implementations of cryptographic algorithms is added based on Elliptic Curve Cryptography (ECC).
- CertPath Algorithm is disabled, where weaker cryptographic algorithms can be disabled.
- Transport Layer Security (TLS1.1/TLS1.2) is supported.
- Endpoint verification is done with the help of the host address and the certificate provided.
- Version number checking during the TLS handshake is stricter.

# Summary

- Cryptography is the mechanism of encoding information in a secret coded form, intended only for the recipient to access the information. Cryptography is used to keep communications limited and private to only the sender and receiver.
- The Java security API is a new addition to library of Java APIs. It is a framework written in Java to access and develop cryptographic functionality and forms part of the Java security API.
- Java Cryptography Extension (JCE) is a set of packages that form a framework. It provides implementations for encryption, key generation and agreement, and Message Authentication Code (MAC) algorithms.
- The Cipher class is one of the core classes from JCE family. It provides the functionality of a cryptographic cipher used for encryption and decryption of data. Encryption of data is achieved in a set of steps.
- Password-Based Encryption (PBE) generates a secret encryption key based on a password provided by the end user. The PBE is used in local file encryption tools ensuring data confidentiality.