Kryptering i Java

Med Java følger et — udvidbart — standardbibliotek til kryptering:

- Java Cryptography Architecture (JCA)

For JCA gælder at:

- Der implementeres generiske klasser og metoder der kan anvendes til kryptering med de medfølgende krypteringsalgoritmer.
- Det er muligt at udvide således at disse metoder kan bruges med vilkårlige eksisterende — og fremtidige — krypteringsalgoritmer.

Java Cryptography Architecture (JCA)

Et Java framework til kryptering, nøglegenerering and nøgleudveksling.

JCA API'erne giver mulighed for:

- Symmetrisk and asymmetrisk kryptering.
- Nøgle-generering, -udveksling, -administration og -opbevaring.
- Message Authentication Codes.
- Digests, certifikater og signaturer.
- Kryptografisk sikre tilfældige tal.

JCA funktionalitet

JCA er udviklet således at det:

- Ikke afhænger af bestemte krypteringsalgoritmer, hashfunktioner, m.m.
- Kan udvides efter behov med nye algoritmer ved hjælp af *CSP*.
- Ikke afhænger af bestemte måder at implementere udvidelser på.
- Ikke afhænger af bestemte måder at anvende JCA på.

Det vil sige, at JCA skal kunne bruges uafhængigt af om man som programmør har behov for an overfladisk anvendelse eller en meget dybdegående anvendelse, såsom:

- Bruge en medfølgende algoritme til kryptering.
- Bruge en algoritme fra en anden udvikler.
- Implementere en ny krypteringsalgoritme.
- Håndtering og udveksling af nøgler.
- m.m.

Cryptographic Service Provider

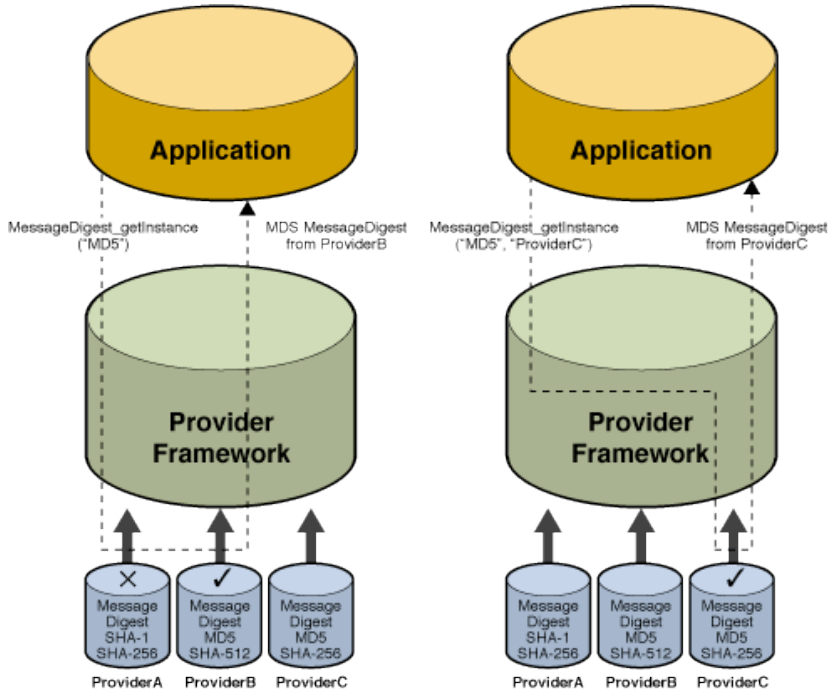
JCA benytter sig af såkaldte "*cryptographic service providers*", der er Java pakker der implementerer kryptografisk funktionalitet, såsom:

- en krypteringsalgoritme
- en hash funktion
- et "key store"
- m.m.

Med JCA følger et antal almindeligt anvendte providers, men der kan udvides med providers fra forskellige udviklere, såsom "The Legion of the Bouncy Castle": <http://www.bouncycastle.org/>, eller man kan selv udvikle sine egne providers.

Terminologien i JCA er at der bruges et antal "*engines*" (implementeret som klasser) der er grænseflader til en specifik funktionalitet (kryptografisk service).

Anvendelse af JCA



http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html

JCA primærklasser (*engines*)

KeyFactory:

- bruges til at konvertere mellem nøgleformater.

KeyPairGenerator:

- bruges til at generere et nøglepar til asymmetrisk kryptering.

KeyStore:

- bruges til at oprette og vedligeholde et "*keystore*", der kan indeholde et antal forskellige nøgler og/eller certifikater.

JCA interfaces

Key:

- interface der giver en generel beskrivelse af en nøgle.

PrivateKey, **PublicKey**:

- interface der giver en generel beskrivelse af henholdsvis en privat og en offentlig nøgle til brug med asymmetrisk kryptering.

SecretKey:

- interface der giver en generel beskrivelse af en hemmelig nøgle til brug med symmetrisk kryptering.

KeySpec:

- interface der giver en generel beskrivelse af informationer der kan bruges til at generere en unik nøgle. Implementeres af et antal forskellige klasser, såsom SecretKeySpec, RSAPrivateKeySpec, …

JCA primærklasser (*engines*)

MessageDigest:

- bruges til at beregne en hash (*message digest*) af angiven data.

Signature:

- bruges til generering og verifikation af en signatur ud fra nøgler og data.

http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html

JCA primærklasser (*engines*)

CertificateFactory:

- bruges til at oprette certifikater og "*Certificate Revocation Lists*".

CertPathBuilder:

- bruges til opbygning af "*certification chains*".

CertPathValidator:

- bruges til validering af "*certification chains*".

CertStore:

- bruges til at hente certifikater og CRL's fra et certifikat-*repository*.

JCA primærklasser (*engines*)

SecureRandom:

- bruges til at generere tilfældige — eller pseudo-tilfældige — tal.

AlgorithmParameters:

- bruges til at håndtere de parametre der bruges ved en bestemt anvendelse af en algoritme — inklusive om det drejer sig om kodning eller afkodning.

AlgorithmParameterGenerator:

- bruges til at generere et nyt AlgorithmParameters objekt til en bestemt anvendelse af en algoritme.

Java Cryptography Extension

En medfølgende udvidelse, der giver adgang til lidt flere af JCA's engines, samt tilføjer en yderligere provider: "SunJCE".

Fokus for JCE er kryptering, hvorfor de engines der nu gives adgang til typisk anvendes ved kryptering.

Oprindeligt blev der lagt vægt på hvad der er JCA og hvad der er JCE, men i de seneste Java versioner glider disse lidt mere sammen, og JCE betragtes nu nærmest som en integreret del af JCA — selv om opdelingen stadig kan ses.

Overblik over JCE klasser (*engines*)

Cipher:

- bruges til kryptering/dekryptering af data — indeholder oplyninger om algoritme, parametre, m.m.

SecretKeyFactory:

- bruges til at konvertere mellem symmetriske nøgleformater.

KeyGenerator:

- bruges til at generere en nøgle til symmetrisk kryptering.

SealedObject:

- indeholder et krypteret objekt.

Mac:

- bruges til generering af Message Authentication Codes.

Java Cryptography Architecture (JCA)

JCA API'et giver mulighed for at håndtere:

- Kryptering.
- Nøgler.
- Sikre, krypterede datastrømme.
- Nøgleudveksling (og protokoller dertil).
- Anvendelse af *cryptographic service providers*.
- Kryptografisk sikre tilfældige tal.
- Message digests, digitale signaturer og certifikater.
- Nøgleadministartion

Eksempel (byte array)

```
import javax.crypto.*;

public class JCEtest {
   public static void main (String args[]) throws Exception {
      byte[] cleartext = "Hello World!".getBytes();
      System.out.println(new String(cleartext));
      KeyGenerator kgen = KeyGenerator.getInstance("AES");
      SecretKey skey = kgen.generateKey();
      Cipher cipher = Cipher.getInstance("AES");
      cipher.init(Cipher.ENCRYPT_MODE, skey);
      byte[] ciphertext = cipher.doFinal(cleartext);
      System.out.println(new String(ciphertext));
      SecretKey badkey = kgen.generateKey();
      System.out.println();
      try {
         cipher.init(Cipher.DECRYPT_MODE, badkey);
         byte[] decrypted = cipher.doFinal(ciphertext);
         System.out.println(new String(decrypted));
      } catch (Exception e) {
         System.out.println("Cannot decipher!");
      };
      cipher.init(Cipher.DECRYPT_MODE, skey);
      byte[] decrypted = cipher.doFinal(ciphertext);
      System.out.println(new String(decrypted));
   }
}
```

Eksempel (`SealedObject`)

```
import javax.crypto.*;
import javax.crypto.spec.*;

public class JCEtest {
   public static void main (String args[]) throws Exception {
      String cleartext = "Hello World!";
      System.out.println(cleartext);
      String skey = "32CharecterSecret = 256BitSecret";
      SecretKeySpec kspc = new SecretKeySpec(skey.getBytes(),"AES");
      Cipher cipher = Cipher.getInstance("AES");
      cipher.init(Cipher.ENCRYPT_MODE, kspc);
      SealedObject so = new SealedObject(cleartext,cipher);
      String badkey = "This must also be 32 characters!";
      SecretKeySpec badspc = new SecretKeySpec(badkey.getBytes(),"AES");
      System.out.println();
      try {
         String s = (String)so.getObject(badspc);
         System.out.println(s);
      } catch (Exception e) {
         System.out.println("Cannot decipher!");
      };
      kspc = new SecretKeySpec(skey.getBytes(),"AES");
      String s = (String)so.getObject(kspc);
      System.out.println(s);
   }
}
```


Provider registrering

Det er muligt at registrere en provider på to måder:

- **statisk**, ved at tilføje providers i en fil: "`$JAVA_HOME/jre/lib/security/Java.security`".
- **dynamisk**, ved at placere provider biblioteket i samme katalog som det program der bruger provideren (og i nogle tilfælde angive en tilføjelse til programmets sikkerhedspolitik).

Eksempel på statisk registrerede providers

```
security.provider.1=sun.security.pkcs11.SunPKCS11 ${java.home}/lib/security/sunpkcs11-macosx.cfg
security.provider.2=sun.security.provider.Sun
security.provider.3=com.apple.crypto.provider.Apple
security.provider.4=sun.security.rsa.SunRsaSign
security.provider.5=com.sun.net.ssl.internal.ssl.Provider
security.provider.6=com.sun.crypto.provider.SunJCE
security.provider.7=sun.security.jgss.SunProvider
security.provider.8=com.sun.security.sasl.Provider
security.provider.9=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.10=sun.security.smartcardio.SunPCSC
```