Session

Introduction to Java Cryptography API

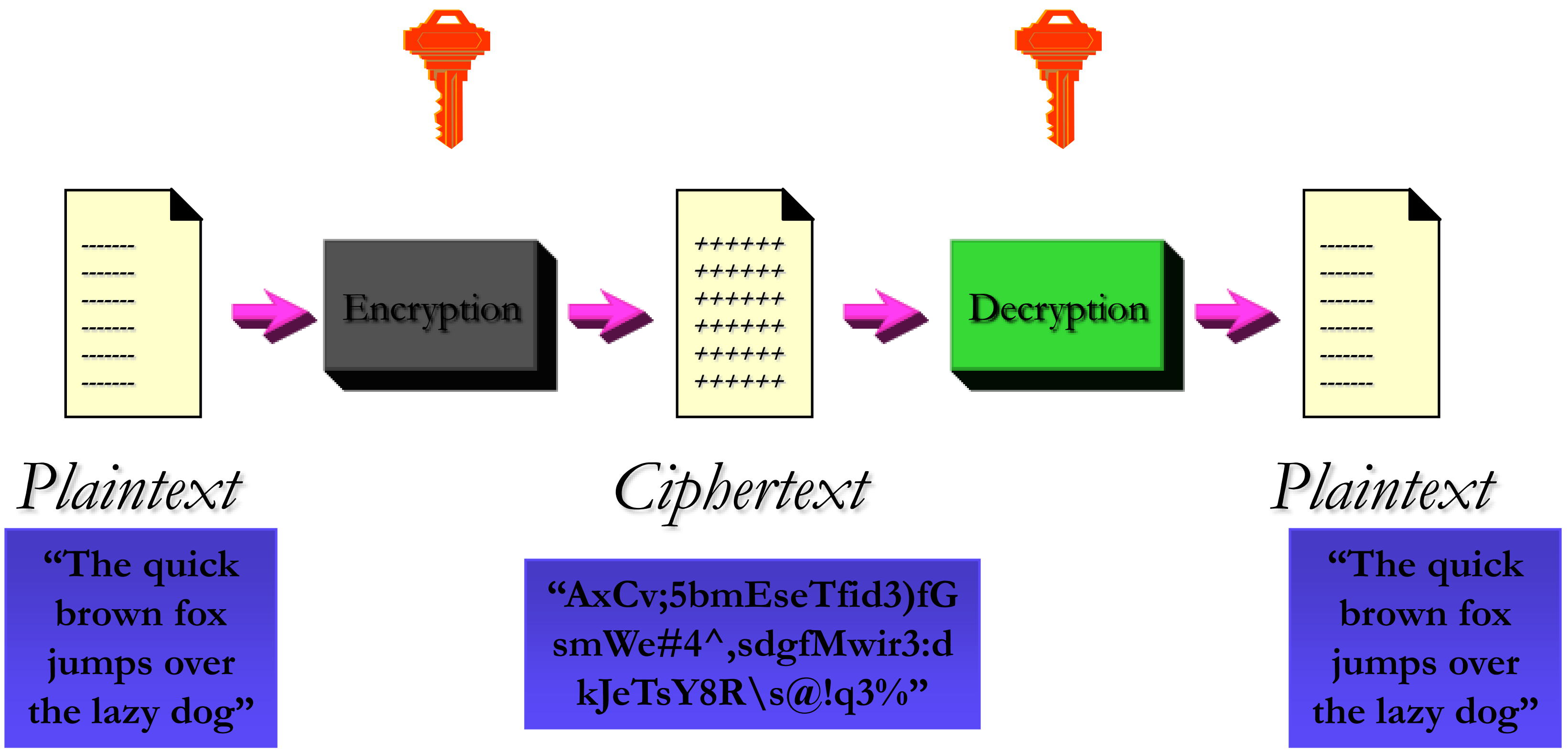
Objectives

- To understand concepts of Java Cryptography API and other APIs in Java

We Should Have Know The Keyword

- **Plaintext**. The message which has to be sent across
- **Cipher**. The encryption/decryption algorithms which are used in the transformation of plaintext to ciphertext and vice-versa
- **Ciphertext**. The message after it is encoded
- **Key**. This is a unique value (bit pattern, alphabetical sequence) that is used by the cipher for encryption/decryption

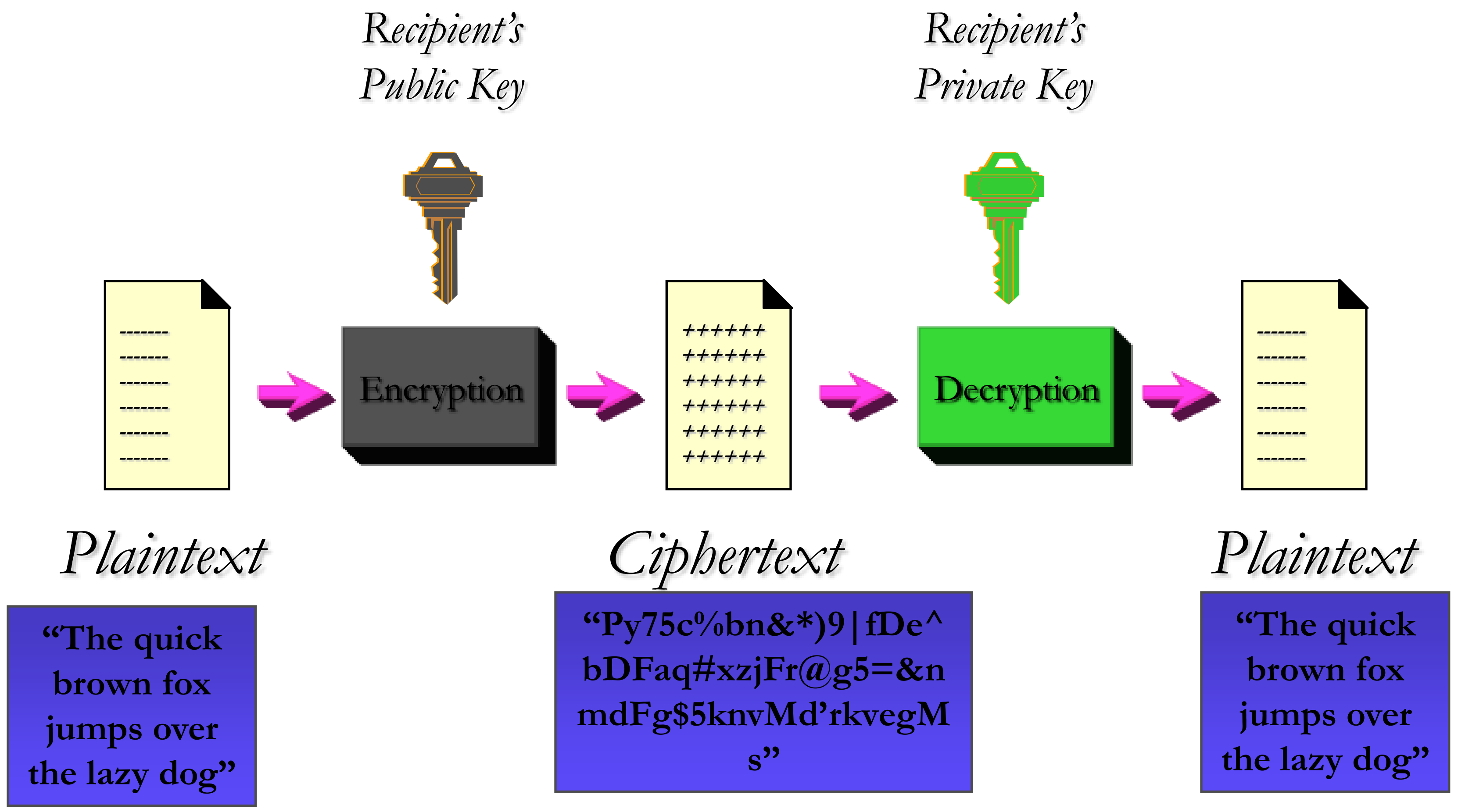
Encryption



Encryption
++++++
++++++
++++++
++++++
++++++
++++++
Decryption

Plaintext
Ciphertext
Plaintext
"The quick brown fox jumps over the lazy dog"
"AxCv;5bmEseTfid3)fGsmWe#4^,sdgfMwir3:dkJeTsY8R\s@!q3%"
"The quick brown fox jumps over the lazy dog"

Public Key Cryptography


Recipient's Public Key
Recipient's Private Key
Encryption
Decryption
Plaintext
Ciphertext
Plaintext
"The quick brown fox jumps over the lazy dog"
"Py75c%bn&*)9|fDe^ bDFaq#xzjFr@g5=&n mdFg$5knvMd'rkvegM s"
"The quick brown fox jumps over the lazy dog"

Java Cryptography Architecture

Introduction: The JCA is a major piece of the platform, and contains a "provider" architecture a set of APIs for digital signatures, digests (hashs),certificates and certificte validation,encryption(symmetric/asymmetric) Key generation and management ,and secure random number generation ,to name a few

These APIs allow developers to easily integrate security into their application code

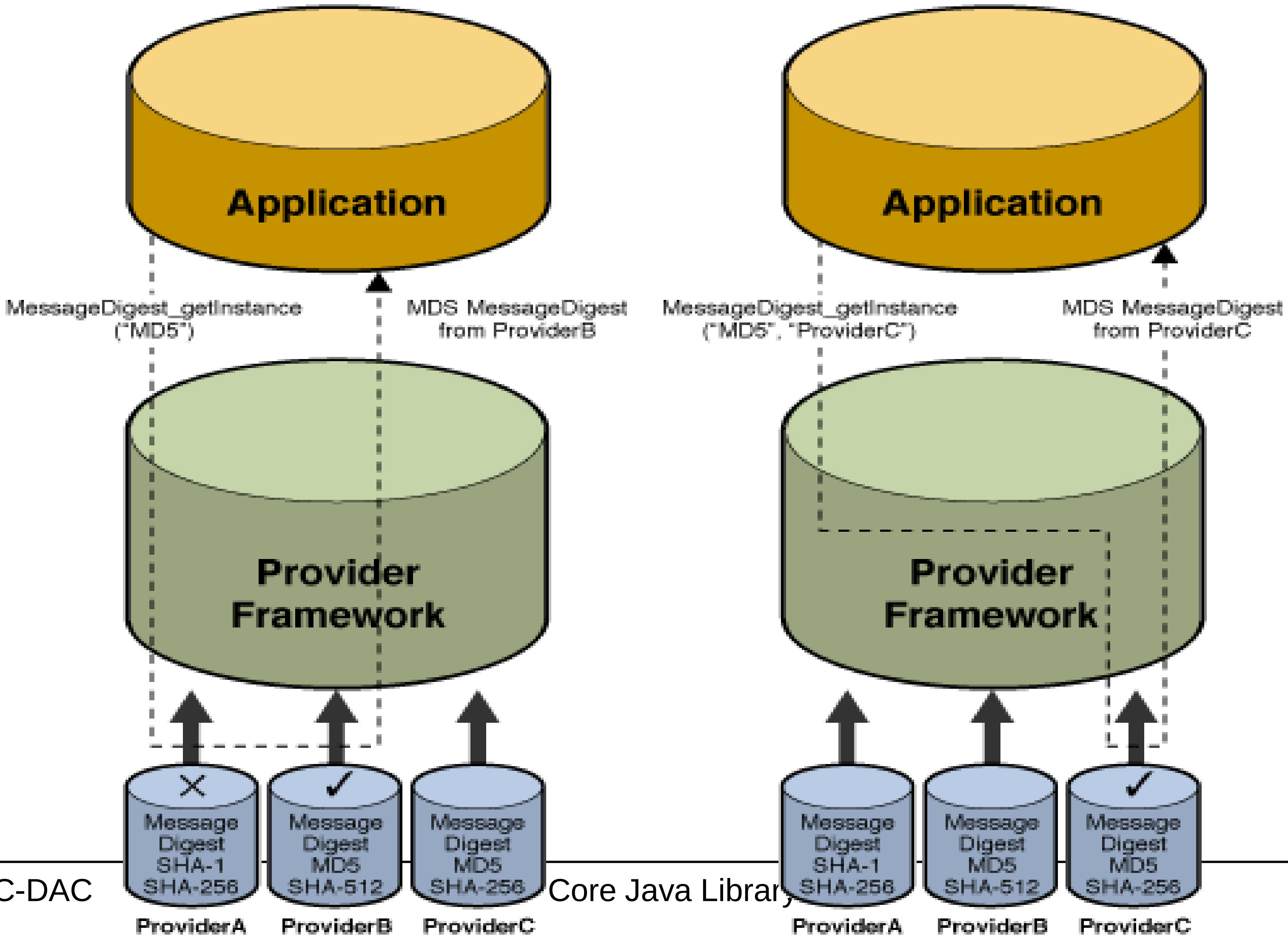
The architecture was designed around the following principles:

Implementation independence: Applications do not need to implement security algo. They request security services from the Provider. Which are plugged into java plateform via a standard Interface , it may be application dependent on multiple providers

Implementation interoperability: provider are interoperable Across application, mean an application is not bound to a specifi Provider , and a provider is not bound to a specific application

Algorithm extensibility : Java plateform includes a no. of built-in Providers that implement a basic set of security services .the java Plateform supports the installation of custum providers.algorithm

JCA Architecture


Application
MessageDigest_getInstance ("MD5")
MDS MessageDigest from ProviderB
Provider Framework
✕
Message Digest SHA-1 SHA-256
ProviderA
✓
Message Digest MD5 SHA-512
ProviderB
Message Digest MD5 SHA-256
ProviderC
Application
MessageDigest_getInstance ("MD5", "ProviderC")
MDS MessageDigest from ProviderC
Provider Framework
Message Digest SHA-1 SHA-256
ProviderA
Message Digest MD5 SHA-512
ProviderB
✓
Message Digest MD5 SHA-256
ProviderC

```
md = MessageDigest.getInstance("MD5");
md = MessageDigest.getInstance("MD5", "ProviderC");
```

java.security.Provider is the base class for all security providers Each CSP contains an instance of this class which contains the provider's name and lists all of the security services/algorithms it implements

Each JDK installation has one or more providers installed and configured by default.

Additional providers may be added staticall or dynamically Clients may configure their runtime environment to specify the provider *preference order*.

Java.Security

Important API under java security

- Java.Security, Java.Security.acl, Java.Security.cer, Java.Security.interfaces, Java.Security.spec
- Javax.crypto, Javax.crypto.interfaces, Javax.crypto.spec

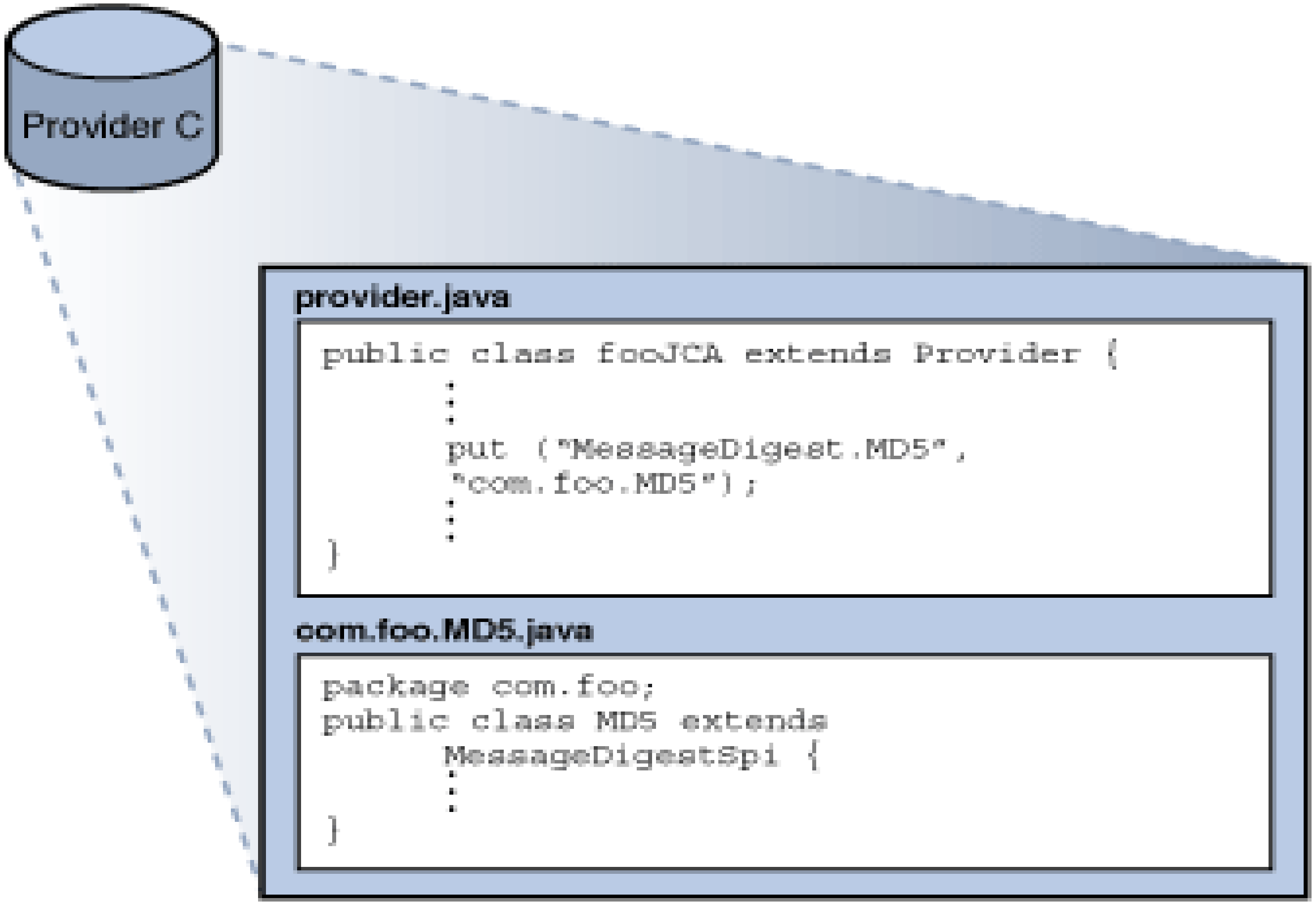
Cryptographic Service Provider refers to a package or set of packages that supply a concrete implementation of a subset of the JDK Security API cryptography features.The Provider *class* is the interface to such a package or set of packages It has methods for accessing the Provider name , version number, and other information

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

```
MessageDigest md = MessageDigest.getInstance("MD5");
KeyAgreement ka = KeyAgreement.getInstance("DH", "SunJCE");
```



```
public class fooJCA extends Provider {
        :
        put ("MessageDigest.MD5",
        "com.foo.MD5");
        :
}
```

```
package com.foo;
public class MD5 extends
        MessageDigestSpi {
        :
}
```

In order to be used, a cryptographic provider must first be installed then registered either statically or dynamically. There are a variety of Sun providers shipped with this release (SUN, SunJCE, SunJSSE, SunRsaSign, etc.) that are already installed and registered.

Installing the Provider Classes: Two possible ways

On the normal Java classpath: Place a zip or JAR file containing the classes anywhere in your classpath.
Some algorithms types (Ciphers) require the provider be a signed Jar file.
As an Installed/Bundled Extension The provider will be considered an *installed* extension if it is placed in the standard extension directory. In Sun's JDK, that would be located in:
<java-home>/lib/ext [Unix] <java-home>\lib\ext [Windows]

Registering the Provider

The next step is to add the provider to your list of registered providers. Providers can be registered statically by editing a security properties configuration file before running a Java application, or dynamically by calling a method at runtime.
To prevent the installation of rogue providers being added to the runtime environment, applications attempting to dynamically register a provider must possess the appropriate runtime privilege.

Static Registration

configuration file is located in the following location:

<java-home>/lib/security/java.security [Unix]
<java-home>\lib\security\java.security [Windows]
For each registered provider, this file should have a statement of the following form: security.provider.*n=masterClassName*
masterClassName must specify the fully qualified name
of provider's master class

How to Generate a Key (Symmetric)

```
import javax.crypto.*;
Import java.math.*;

class KeyGen {
   public static void main(string []args) throws Exception {
     KeyGenerator kg = KeyGenerator.getInstance("DES");
     SecretKey key = kg.generateKey();
      BigInteger b = new BigInteger(1, key.getEncoded());
      System.out.println(b.toString(16));

   }
}

//Outputs will vary each time; as a new key will be
   generated;
//AES is another symmetric key algorithm
```

Encryption (Symmetric Key)

- Cipher Objects and methods

```
Cipher.getInstance(“DES/ECB/PKCS5Padding”);

Cipher.init(Cipher.ENCRYPT_MODE,secretkey);

Cipher.doFinal(plaintext);
```

Decryption (Symmetric Key)

- Cipher Objects
  - **`Cipher.init(Cipher.DECRYPT_MODE,secretkey);`**
  - **`Cipher.doFinal(ciphertext);`**

Example – Symmetric Key

```
import java.io.*;
import java.security.*;
import javax.crypto.*;

public class Example {
    public static void main(String args[]) throws IOException{
     try       {
           byte[] plainText = args[0].getBytes("UTF8");

           // Create Key
           KeyGenerator kg = KeyGenerator.getInstance("DES");
           SecretKey secretKey = kg.generateKey();
           System.out.println("Key:"+ new BigInteger(1,
                  secretKey.getEncoded()).toString(16));

           // Create Cipher
Cipher desCipher =Cipher.getInstance("DES/ECB/PKCS5Padding");
desCipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

Example.

```
      // Encrypt the plaintext using the secret key
System.out.println("Start Encryption");
byte[] cipherText = desCipher.doFinal(plainText);
System.out.println( "Finish encryption:" );
System.out.println( new String(cipherText, "UTF8") );

      // Initializes the Cipher object.
desCipher.init(Cipher.DECRYPT_MODE, secretKey);

    // Decrypt the ciphertext using the same secret key
System.out.println("Start Decryption") ;
byte[] NewPlainText = desCipher.doFinal(cipherText);
System.out.println( "Finish decryption: " );
System.out.println( new String(NewPlainText, "UTF8") );
}
  catch(Exception e){ System.out.println("Error");}
}}
```

Example – 2

```
import javax.crypto.Cipher;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import java.security.Key;
import java.security.InvalidKeyException;
import java.util.Arrays;
import java.math.*;

public class SymmetricCiphers {
   private static String algorithm = "DESede"; //AES
   private static Key key = null;
   private static Cipher cipher = null;
```

Example – 2 – Continued …

```
private static void setUp() throws Exception {
  key = KeyGenerator.getInstance(algorithm).generateKey();
  cipher = Cipher.getInstance(algorithm);
}

public static void main(String[] args) throws Exception {
  setUp();
  byte[] encryptionBytes = null;
  String input = "Hello.";
  System.out.println("Input Text: " + input);
  System.out.println("Key: "+ new BigInteger(1,
   key.getEncoded()).toString(16));
  encryptionBytes = encrypt(input);
  System.out.println("Encrypted Text:" + new BigInteger(1,
   encryptionBytes).toString(16));
  System.out.println("Decrypted Text:" +
   decrypt(encryptionBytes));
}
```

Example – 2 – Continued …

```
private static byte[] encrypt(String input)throws
   InvalidKeyException, BadPaddingException,
   IllegalBlockSizeException {
     cipher.init(Cipher.ENCRYPT_MODE, key);
     byte[] inputBytes = input.getBytes();
     return cipher.doFinal(inputBytes);
}
private static String decrypt(byte[] encryptionBytes)
   throws InvalidKeyException, BadPaddingException,
   IllegalBlockSizeException {
   cipher.init(Cipher.DECRYPT_MODE, key);
    byte[] recoveredBytes = cipher.doFinal(encryptionBytes);
    String recovered = new String(recoveredBytes);
    return recovered;
}
```

Symmetric Key Cryptography

- Symmetric approach uses one key for encryption & decryption.
- Example you just saw was Symmetric. Same key was used in encryption & decryption.
- Key transfer is problem over here.

Asymmetric Key Cryptography

- Asymmetric System uses two different keys.
  - Public key
  - Private key
- One for encryption & other for decryption.
- We require to do the following things.
  - Generate Public key
  - Generate Private key
  - Encrypt message with Public key
  - Decrypt message with Private key

Generating Public & Private Keys

- `KeyPairGenerator.getInstance(“RSA”)`
- `KeyPairGenerator.initialize(1024)`
- `KeyPairGenerator.generateKeyPair()`
- `KeyPair`
  - `KeyPair.getPublic()`
  - `KeyPair.getPrivate()`

Example - Asymmetric Key Gen

```
import java.io.*;
import java.security.*;
import javax.crypto.*;

class example2 {
   public static void main (String args[]) {
        try{
                // Generate an RSA key
            System.out.println( "\nStart generating RSA key" );
            KeyPairGenerator keyGen =
      KeyPairGenerator.getInstance("RSA");
            keyGen.initialize(1024);
            KeyPair key = keyGen.generateKeyPair();

        System.out.println( "Finish generating RSA key" );
        System.out.println( "Private Key :" + key.getPrivate());
        System.out.println("Public Key :" +   key.getPublic());
   }
        catch(Exception e){System.out.println("ERROR");}
}}
```

Encryption & Decryption Example.

```
public static byte[] encrypt(String text, KeyPair
  key)
{
  byte[] cipherText = null;
  try {
   final Cipher cipher = Cipher.getInstance("RSA");
  cipher.init(Cipher.ENCRYPT_MODE, key.getPublic());
  cipherText = cipher.doFinal(text.getBytes());
  }
  catch (Exception e) { e.printStackTrace(); }
  return cipherText;
}
```

Message Digest

- Java.security.MessageDigest
- MessageDigest.getInstance(“MD5”);
- MessageDigest.update(plain text);
- MessageDigest.digest()

Message Digest Example

```
import java.security.*;
import java.math.*;

public class Message {
   public static void main (String[] args)throws Exception {
     MessageDigest md = MessageDigest.getInstance("MD5");
     String text = “password123”;
     byte[] plainText = text.getBytes("UTF8");
     md.update(plainText); //updates input to Digest
     byte[] mdbytes = md.digest(); //completes hash operation
     BigInteger b = new BigInteger(1, mdbytes);
     System.out.println(“Message Digest=“ + b.toString(16));
   }
}
```

Output:

```
Message Digest = 482c811da5d5b4bc6d497ffa98491e38
```

//SHA-1, SHA-256, SHA-384, SHA-512 are other algorithms supported

Digital Signature

- Java.security.Signature
- Generating Signature:
  - Signature.getInstance(“SHA1withDSA”,”SUN”);
  - Signature.initSign(private key);
  - Signature.update(Message, length);
  - Signature.sign();
- Verify Digital Signature:
  - Signature.getInstance(“SHA1withDSA”,”SUN”);
  - Signature.initVerify(public Key);
  - Signature.update(Message, length);
  - Signature.verify(Signature);

Digital Signature - Example

```
import java.io.*;
import java.security.*;
import javax.crypto.*;

public class DigitalSigning {
   KeyPair key;
   Signature signer;
   String data = "Hello!! Don't share this secret!";
   byte[] dataToSign;

   void generateKeys()    {
        try {
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
         keyGen.initialize(1024);
         key = keyGen.generateKeyPair();
        } catch (Exception e) {}
   }
```

Digital Signature – Example

```
byte[] Sign()       {
  byte[] signeddata = null;
  try {
       signer = Signature.getInstance("SHA1withRSA");
       signer.initSign(key.getPrivate());
       dataToSign = data.getBytes();
       signer.update(dataToSign);
       signeddata = signer.sign();
      }  catch (Exception e) {}
   return signeddata;
}
public static void main(String []args)   {
   DigitalSigning ds = new DigitalSigning();
   ds.generateKeys();
    byte[] signeddata = ds.Sign();
```

Digital Signature – Example

```
System.out.println("Data to be signed = "+ds.data);
BigInteger b = new BigInteger (1,
   ds.key.getPrivate().getEncoded());
System.out.println("PrivateKeyUsed=" + b.toString(16));
b = new BigInteger(1, signeddata);
System.out.println("Signed Data = " + b.toString(16));
}}
```

Digital Signature – Output

```
Data to be signed = Hello!! Don't share this secret!
Private Key Used =
   3082027602010030d06092a864886f70d0101010500048202603082025c02010002818100e32e
   ba520ad8ccd3ed35c5fdaf992d36243ff9673002ccff040ddb0a81c0c7c7e2572e92fd96c63a6b
   33e234ba24672b90ad74991d38565f0ca3601f1fbf2497ef979f0797d70e4aa0400c6903672514
   3b2309afe6c9a5e23235542f103fb367c0f3fe5574d79d0cecd84e754495c39ff19eca5442fa7f
   601319c31f3041045b020301000102818 02ba57b986d8b0f771bd8bbec2c436283475d9f1deb04
   6ebc03dc619bd827e07a0f8b2e608125e979a9c4f30dac41782ccf83f3a5c30e62d1aa0ced57ad
   72bca5b365d805fc0a1d503e794a6c38f0676267914a9ec9c4b15aecc7552eb69469a6c5d007ce
   7d668d0f6150c77e5de808e065e1137d66b449510814dc806a60ba69024100f8a84796dbef924f
   063c026ae244eb055aa960a090f5ff3950478aad319d481ddbdb59823a27107f746c3fd36db9a8
   584bf16f62909a015160f6bd078205b1ef024100e9e41c1844acebc7f67455ada0b443cd102e24
   c210d9d4554652acdbcd885e7efe1bcd94856ddf000dc711267f528b768471f00c9790bd6293d8
   264af0a2105502410 0ac3b532ae9382dad52f229f282bb9dd65d8fc8802f28551a0bc3220908be
   9a7f2413f111c4d9de118a40988d08097ad37df6c362102abc12f408b3b2099b8c3d02401572df
   c8b1f391a3c2cc245749d77e283e059d4556ba432896cc5a21c6156d6503f494c3bc00b9648dde
   e589bc3f5b9ec0c29a1aed834e7acdc812aae8aa540902403300d52e5f406e71254b2e3b7f3b91
   4e014f5e3d6472a02874ea35ef4fb2648e82c51b42401036b14e45b3d6f9d17831c30ec1257b84
   8e45351540b428e56797
Signed Data =
   83dbec624cdacfc24f15771797cbb2a8e37520d70a44eee5f6e2bcf8113b01948c50ee1ceb4f17
   42060e9cc999325653904d405a9ac54e6b68e40eba9f47afbd6ccc1ab21ed4517ccf8dcc1346f2
   c8a8614a935f5c2b8fed9fd1cf4256ac3b099e3a15ebfb4263c1ab808e491bbc31bc3957311a14
   8190506be6ba2a7d93daf0
```

Digital Certificate

- Java.seurity.cert
  - Java.security.cert.Certificate
  - Java.security.cert.X509Certificate
  - Java.security.cert.CertificateFactory
  - Java.security.cert.CertStore
- Types of Certificate:
  - der
  - pem

Digital Certificate (cont.)

- CertificateFactory.getInstance(“X509”);
- CertificateFactory.generateCertificate(Cert.der)
- X509Certificate.getPublicKey()
- X509Certificate.getIssuerDN()
- X509Certificate.verify(public key)
- X509Certificate.getSignature()
- X509Certificate.getSigAlgoName()

Certificate Revocation List (CRL)

- Java.security.cert
  - X509CRL
  - X509CRLEntry
  - X509CRLSelectors
  - X509
  - CertificateFactory

- Implementing CRL
  - CertificateFactory.getInstance(“X509CRL”);
  - CertificateFactory.generteCRl(crlFile)

CRL (Cont.)

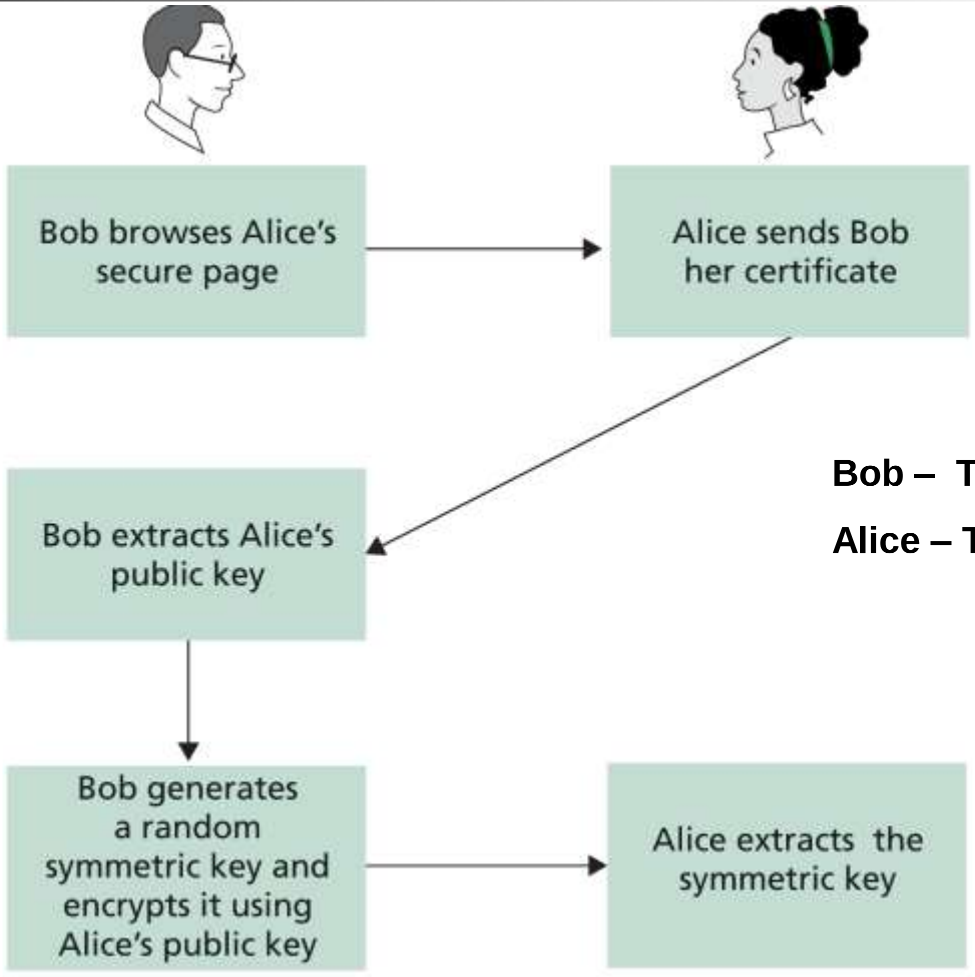
- Checking for Revoked Certificates
  - X509.getRevokedCertificates()
  - X509.getRevokedCertificate(Serial Number)
  - X509.getRevokedCertificate(Certificate)

- Other Methods
  - X509CRL.getNextUpdate()
  - X509CRL.getThisUpdate()
  - X509CRLEntry.getRevokationDate()

SSL

- SSL (Secure Sockets Layer)
  - Originally developed by Netscape
  - Provides Transport Layer Security (HTTPS is HTTP over SSL)
  - Version 1.0 never released; 2.0 released in 1995 – with lot of flaws; 3.0 released in 1996;
  - Basic Algorithm written by Dr. Taher Elgamal – father of SSL
  - Uses X.509 certificates to authenticate the counterparty and exchange a symmetric key
    - Due to X.509, you need CA and PKI in place to generate, sign, and administer the validity of the certificates

How SSL Works?


Bob browses Alice's secure page
Alice sends Bob her certificate
Bob extracts Alice's public key
Bob generates a random symmetric key and encrypts it using Alice's public key
Alice extracts the symmetric key

Bob – The User @ Client / Browser

Alice – The Merchant @ Server

Courtesy: James F Kurose, Keith W Ross, Computer Networking: A Top-Down approach featuring the Internet

SSL

- Provides a secure channel between communicating devices on the net
- SSL is a protocol in the network protocol stack. It resides between the application and the TCP/IP protocols (illustrated in the next slide)
- SSL uses both RSA and Diffie-Hellman Algorithms
- In theory SSL can be used by any application level protocol but at the moment it is used for securing HTTP transactions

SSL (Secure Sockets Layer)

What is provided by SSL

- Confidentiality (privacy)
- Data Integrity (Tamper Proofing)
- Server Authentication (Proving a server is what it claims it is)
- Used in typical B2C transaction
- Optional Client Authentication would be required in B2B (or web services environment in which program talks to program)

SSL

- SSL Certificates are typically issued to Fully-qualified domain names (Eg: webmail.cdac.in, mail.google.com)
  - However Wildcard SSL Certificates can also be obtained (Eg: *.google.com)

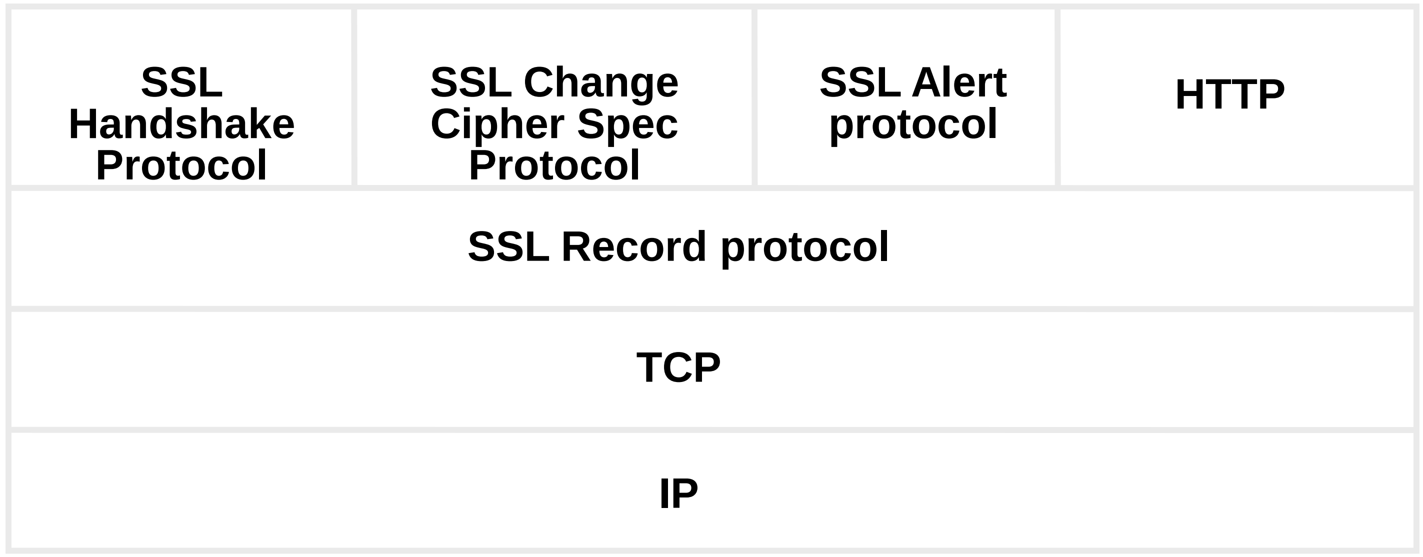
SSL

- SSL Server Authentication
  - SSL-enabled client can use PKC to check that the server’s certificate and public ID are valid, and that the CA is trusted
- SSL Client Authentication
  - SSL-enabled server can check that a client’s certificate and public ID are valid, and that the CA is trusted
- Secure connection – client/server transmissions are encrypted, plus tamper detection

SSL

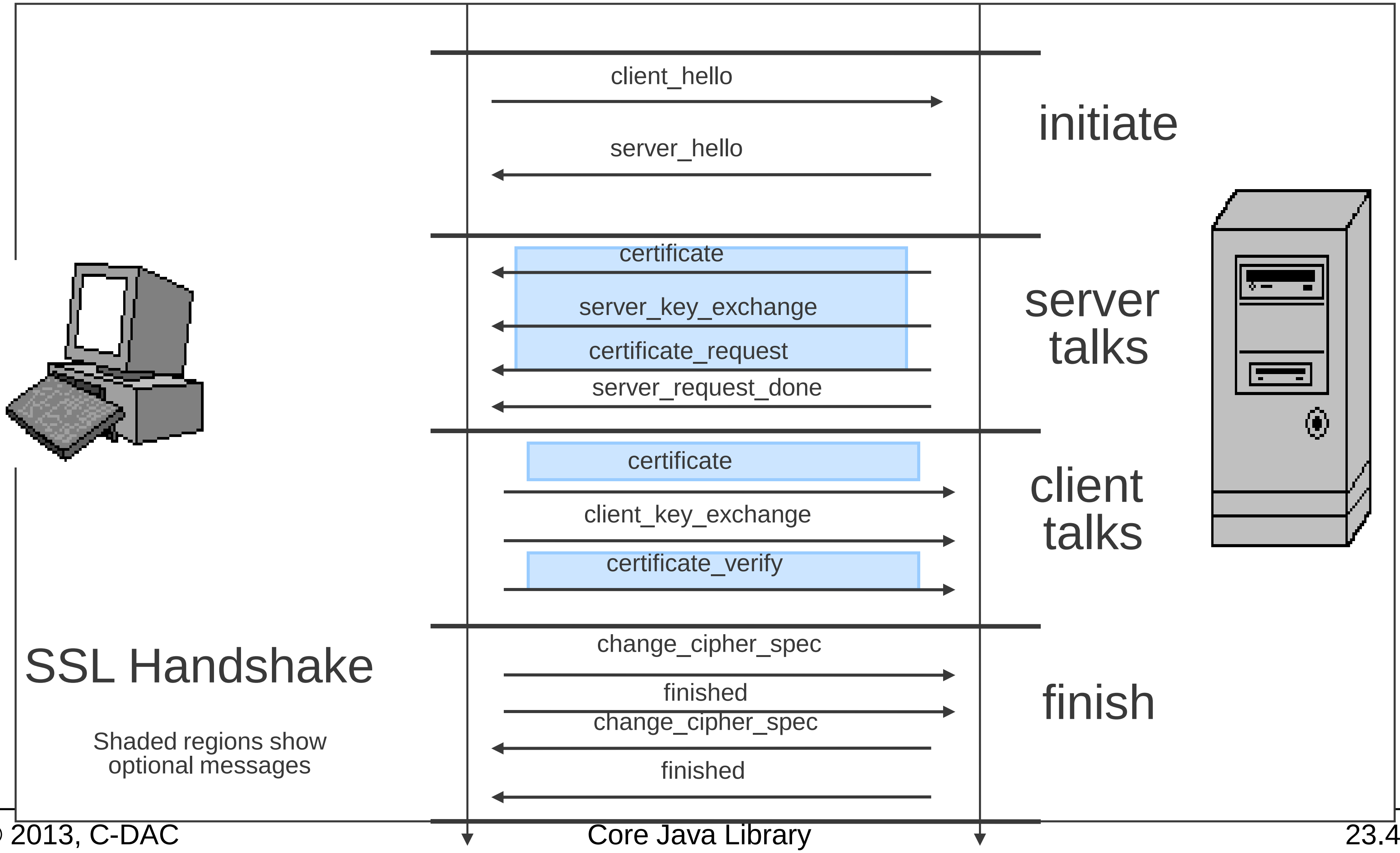
- SSL exchanges messages that permit:
  - client to authenticate the server (always)
  - server to authenticate the client (optional)
  - client and server negotiation of crypto algorithms that they both support
  - using PKC to encrypt and exchange shared secrets
  - establishing an encrypted SSL connection

SSL Protocol Stack



- SSL is actually a suite of protocols of which the record and the handshake protocols are the key ones.
- The handshake protocol is used to set up the session.
- The record protocol is used to receive/transmit the data passed to it from the other sub-protocols (including the handshake protocol

SSL Handshake Protocol



TLS

- TLS is an IETF Standards track protocol
  - First defined in 1999 (RFC 2246) as an upgrade to SSL 3.0
  - Updated in 2008 (RFC 5246) and 2011 (RFC 6176)
  - Based on the SSL Specifications developed by Netscape Communications; and uses the same X.509 certificates
    - X.509 Certificates
      - Standard by ITU-T
      - Formats for Public Key Certificates

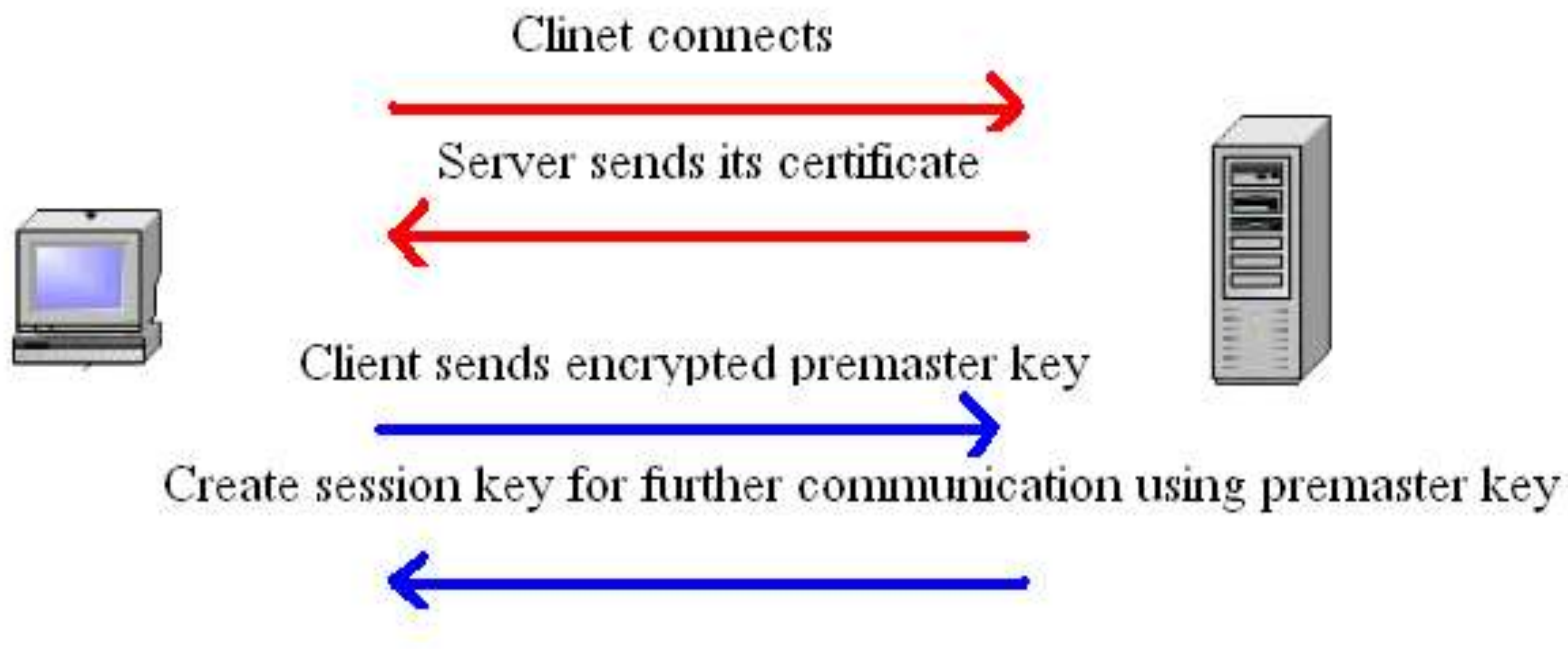
SSL and Security Keys

-Uses public/private key (asymmetric)
-Scheme to create secret key (symmetric)

-Secret key is then used for encryption of data
-SSL operation is optimized for performance:

-Using symmetric key for encryption is a lot faster than using asymmetric keys

SSL Key Exchange


Clinet connects
Server sends its certificate
Client sends encrypted premaster key
Create session key for further communication using premaster key

SSL Authentication

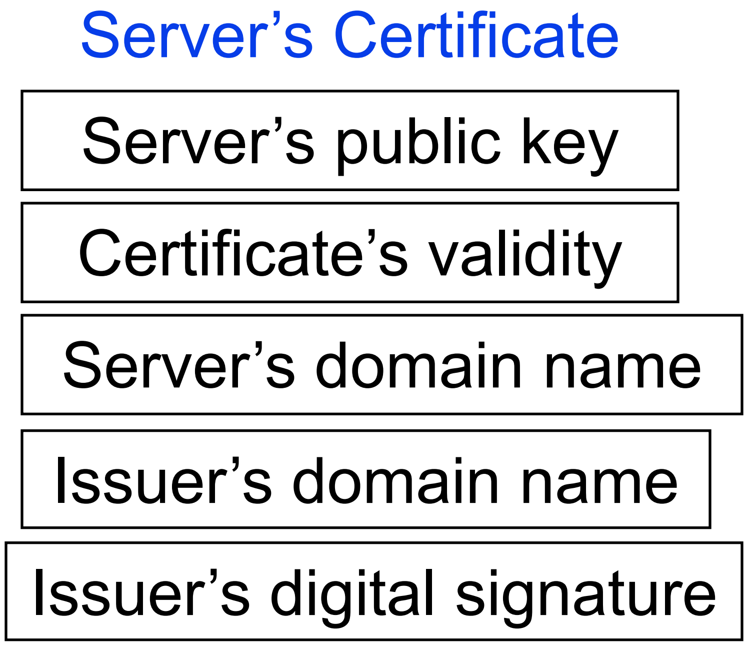
1. For server authentication, the client encrypts the premaster secret with the server's public key.

2. Only the server’s private key could have decrypted that data.

3. For client authentication, client encrypts some data known to client and server with client’s private key (i.e., creates a digital signature).

Public key in client’s certificate will validate the digital signature only if it was encrypted with the client’s private key.

Server Authentication


Server's Certificate

1. Is today's date within validity period?
2. Is issuing CA a trusted CA?
3. Does issuing CA's public key validate the issuer's digital signature?
4. Does the domain name in the server's certificate match the domain name of the server itself?

Programming - Server

```
import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLSocket;

SSLServerSocketFactory sssf =
(SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();

SSLServerSocket sss = (SSLServerSocket)
sssf.createServerSocket(9999);

SSLSocket ss = (SSLSocket) sss.accept();

InputStream inputstream = sslsocket.getInputStream();
..
```

Programming - Client

```
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;

SSLSocketFactory ssf = (SSLSocketFactory) ssf.getDefault();

SSLSocket ss = (SSLSocket) ssf.createSocket("localhost",
                                                  9999);

OutputStream outputstream = sslsocket.getOutputStream();
..
```

Summary

- Java Cryptography APIs
- SSL