Symmetric versus asymmetric cryptography. In symmetric the encryption and decryption keys are the same while in asymmetric cryptography they are different.

Public key cryptography. (asymmetric)

RSA scheme. (asymmetric)

Examples: Secure shell (ssh, slogin, sshd), PGP (Pretty Good Privacy, a free cryptographic package), Kerberos network authentication, SSL (Secure Sockets Layer, used with https protocol).

“If privacy is outlawed, only outlaws will have privacy.” Zimmerman (author of PGP)
Public key cryptography

Each participant has a *public key* and a *secret key*. In RSA public-key cryptosystem, each key consists of a pair of large integers.

*Alice* has key \((P_A, S_A)\).

*Bob* has key \((P_B, S_B)\).

Let \(D\) be the set of permissible messages. Then we require the following conditions.

\[
P_A, S_A, P_B, S_B : D \rightarrow D
\]

\[
M = S_A(P_A(M))
\]

\[
M = P_A(S_A(M))
\]

\[
M = S_B(P_B(M))
\]

\[
M = P_B(S_B(M))
\]
Sending an encrypted message

1. Bob obtains Alice’s public key $P_A$.
2. Bob computes the ciphertext $C = P_A(M)$ corresponding to the message $M$ and sends $C$ to Alice.
3. When Alice receives the ciphertext $C$, she applies her secret key $S_A$ to retrieve the original message: $M = S_A(C)$. 
Digital signature

1. Alice computes her digital signature $\sigma = S_A(M')$ for the message $M'$ using her secret key.
2. Alice sends the message/signature pair $(M', \sigma)$ to Bob.
3. When Bob receives $(M', \sigma)$, he can verify that it originated from Alice by using Alice's public key to verify that $M' = P_A(\sigma)$.

A digital signature is verifiable by anyone who has access to the signers public key. The signed message is not encrypted.
Encrypted and signed message

1. Alice computes her digital signature $\sigma = S_A(M')$ for the message $M'$ using her secret key.
2. Alice appends her digital signature to the message and then encrypts the resulting pair with Bob’s public key to send $P_B(M'\sigma)$.
3. Bob decrypts the message using his secret key.
4. Bob verifies Alice’s signature using her public key.
More on cryptography

- The security of the public-key cryptosystem rests in large part on the difficulty of factoring large integers.
  - If factoring large integers is easy, then breaking the RSA cryptosystem is easy.
  - If factoring large integers is hard, then whether breaking RSA is hard is an unproven statement. However decades of research has not found an easy way to break the RSA system.

- A perfect tool for electronic contracts, electronic checks, e-cash, etc. However cryptography is not a panacea for every security issue.

- How do you get your public key in the beginning. Get a certificate from a trusted authority.

- Public-key cryptosystem involve multiple-precision arithmetic which is considerably slower. Most practical systems use a hybrid approach.
Encryption in Java

- The packages `javax.crypto` and `java.security` provide the basic mechanisms.
- We can get a list of all available security providers with some simple code. See example: `security/providers/ListProviders.java`
- See code snippet below:

```java
import java.security.Provider;
import java.security.Provider.Service;
import java.security.Security;
import java.util.Set;

public class ListProviders {
    public static void main (String[] args) {
        Provider[] list = Security.getProviders();
        for (Provider e: list) {
            System.out.println(e);
            Set<Service> serviceList = e.getServices();
            for (Service s: serviceList)
                System.out.println("\t" + s);
            System.out.println();
        }
    }
}
```
Encryption in Java

- Sample code that generates a key and uses it to encrypt information and then stores both the key and the encrypted information on the disk. See examples:
  - EncryptTest.java
  - DecryptTest.java
- Uses the Advanced Encryption Standard (AES)
- Code snippet:

```java
// security/Encryption/EncryptTest.java
FileOutputStream dataFile = new FileOutputStream("data.encrypted");
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("key");

KeyGenerator kg = KeyGenerator.getInstance("AES");
Key key = kg.generateKey();
oos.writeObject(key);

Cipher cipher = Cipher.getInstance("AES");
byte[] data = "Hello World!".getBytes();
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] result = cipher.doFinal(data);
dataFile.write(data);
```

- **Beware!** The key is not protected. It must at least not be readable by anyone else.
Decryption in Java

Sample code that reads in a key and uses it to decrypt information from an encrypted file.

```java
// security/Encryption/DecryptTest.java
File dataFile = new File("data.encrypted");
FileInputStream data = new FileInputStream(dataFile);
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("key"));

Key key = (Key) ois.readObject();
Cipher cipher = Cipher.getInstance("AES");
byte[] result = new byte[(int)dataFile.length()];
int n = data.read(result);
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] original = cipher.doFinal(result);
```
See example securehash/SHA2Test.java on how to use MD5 sums and Secure hash functions in Java. Please note that MD5 sums have a known flaw so research them fully before relying on them.

Another issue is handling password input without showing input. See the example PasswordTest.java in the folder security/passwordinput for more details.
Secure Sockets Layer

- **Secure Sockets Layer (SSL)**. The most widely used protocol used for SSL provides privacy, data integrity, authenticity and non-repudiation.
  - Uses asymmetric key cryptography (public/private key) to authenticate the identities of the communicating parties.
  - Uses asymmetric key cryptography (public/private key to encrypt the shared encryption key that is used during the SSL session.
  - Uses symmetric key cryptography for data encryption between client and server.

- **RMI with SSL**. Java has two classes that provide the ability to secure the communication channel using SSL/TLS (Secure Socket Layer/Transport Layer Security) protocols:
  - `javax.rmi.ssl.SslRMIClientSocketFactory`
  - `javax.rmi.ssl.SslRMIServerSocketFactory`
Overview of RMI with SSL

- _Establish a SSL session using a SSL handshake_. The client initiates a connection and the server responds. This a multi-step process. The net result is that the client and server agree on an encryption scheme.

- _The server sends its certificate and the client verifies it_. The certificate sent by the server comes from the server’s keystore as a database for the contents. The client either verifies the server’s signature by either trusting the server or trusting one of the signers in the certificate chain provided by the server.

- _The client stores certificates in its truststore_. The default truststore is cacerts, which can be found in the following location: $JAVA_HOME/jre/lib/security/cacerts. Check the listed authorities in it with the command (default password is changeit):

  ```
  keytool -list -v -keystore cacerts
  ```

- _Next the client uses the public key of the server to send a ClientKeyExchange message to the server_. The message contains some random information that is used to generate a symmetric key that will be used for encrypting the content during the data exchange.

- _Next the client sends a ChangeCipherSpec message indicating that it is ready to communicate_. This message is followed by a Finished message.

- _The server responds by sending its ChangeCipherSpec message and a Finished message_.


Example SSL Handshake

*** ClientHello, TLSv1
Session ID: {}
Cipher Suites: [SSL_RSA_WITH_RC4_128_MD5, SSL_RSA_WITH_RC4_128_SHA, TLS_RSA_WITH_AES_128_CBC_SHA, TLS_DHE_RSA_WITH_AES_128_CBC_SHA, TLS_DHE_DSS_WITH_AES_128_CBC_SHA, SSL_RSA_WITH_3DES_EDE_CBC_SHA, SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA, SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA, SSL_RSA_WITH_DES_CBC_SHA, SSL_DHE_RSA_WITH_DES_CBC_SHA, SSL_DHE_DSS_WITH_DES_CBC_SHA, SSL_RSA_EXPORT_WITH_RC4_40_MD5, SSL_RSA_EXPORT_WITH_DES40_CBC_SHA, SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA, SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA]
Compression Methods: { 0 }
***

*** ServerHello, TLSv1
Cipher Suite: SSL_RSA_WITH_RC4_128_MD5
Compression Method: 0
***
Example SSL Handshake (continued)

From Client
*** ClientKeyExchange, RSA PreMasterSecret, TLSv1
Random Secret:  { 3, 1, 155, 121, 164, 80, 202, 181, 110, 118, 28, 78, 85, 173, 230, 166, 234, 188, 171, 204, 130, 167, 6, 155, 155, 178, 70, 20, 88, 244, 141, 220, 177, 167, 4, 147, 24, 129, 165, 171, 70, 23, 132, 74, 144, 20, 156, 227 }

From Client
main, WRITE: TLSv1 Change Cipher Spec, length = 1
[Raw write]: length = 6
0000: 14 03 01 00 01 01

*** Finished

From Server
RMI TCP Connection(2)-132.178.248.51, WRITE: TLSv1 Change Cipher Spec, length =
[Raw write]: length = 6
0000: 14 03 01 00 01 01

*** Finished

Generated using the -Djavax.net.debug=all option to the java VM.
Creating the setup

- Generate a keystore that has a key pair (public and private key) along with a self-signed certificate.
  ```
  keytool -genkey -alias SecureServer -keyalg RSA -keystore Server_Keystore
  ```
- Examine the contents of the generated Server Keystore.
  ```
  keytool -list -v -keystore Server_Keystore
  ```
- Create a self-signed certificate.
  ```
  keytool -export -alias SecureServer -keystore Server_Keystore -rfc -file Server.cer
  ```
- To see what the certificate looks like.
  ```
  cat Server.cer
  ```
- Next we import the server certificate into a truststore that can be used by the client.
  ```
  keytool -import -alias SecureServer -file Server.cer -keystore Client_Truststore
  ```
- To verify the contents of the truststore that we created, we issue the following command.
  ```
  keytool -list -v -keystore Client_Truststore
  ```

In our example, we are working with a self-signed certificate instead of certificates signed by Certification Authority (CA). If there is a need to get the certificate signed by a CA then a Certificate Signing Request (CSR) needs to be generated. The generated CSR, then, should to be submitted along with other pertinent information to a Certification Authority such as VeriSign or USPS, who will then digitally sign the certificate.
Using SSL Sockets

- See example in the folder `security/sslsockets`.
- On the server side.

```java
import javax.net.ssl.*;
System.setProperty("javax.net.ssl.keyStore", "Server_KeyStore");
System.setProperty("javax.net.ssl.keyStorePassword", "password");
SSLSocketFactory sslSrvFact =
    (SSLSocketFactory) SSLServerSocketFactory.getDefault();
SSLServerSocket ss = (SSLServerSocket) sslSrvFact.createServerSocket(port);
SSLSocket sock = (SSLSocket) ss.accept();
```

- On the client side.

```java
import javax.net.ssl.*;
System.setProperty("javax.net.ssl.trustStore", "Client_Truststore");
System.setProperty("javax.net.ssl.trustStorePassword", "password");
SSLSocketFactory sslFact = (SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket server = (SSLSocket)sslFact.createSocket(host, port);
```
Using RMI with SSL Sockets

- See examples in the folder `security/rmisslex1`
- On the server side.
  ```java
  System.setProperty("javax.net.ssl.keyStore","./resources/Server_Keystore");
  System.setProperty("javax.net.ssl.keyStorePassword","password");
  System.setProperty("java.security.policy","./resources/mysecurity.policy");
  RMIClientSocketFactory rmiClientSocketFactory =
      new SslRMIClientSocketFactory();
  RMIServerSocketFactory rmiServerSockeyFactory =
      new SslRMIServerSocketFactory();
  DateServer server = (DateServer) UnicastRemoteObject.exportObject(this, 0,
      rmiClientSocketFactory, rmiServerSockeyFactory);
  Registry registry = LocateRegistry.createRegistry(port);
  registry.rebind(name, server);
  ```
- On the client side.
  ```java
  System.setProperty("javax.net.ssl.trustStore", "Client_Truststore");
  System.setProperty("java.security.policy", "mysecurity.policy");
  Registry reg = LocateRegistry.getRegistry(host, port);
  DateServer server = (DateServer) reg.lookup("DateServerImpl");
  ```
- Also see example in folder `security/rmisslex2`
References

- JSSE (Java Secure Sockets Extension) documentation.